IDA PAPER P-2108

Ada LEXICAL ANALYZER GENERATOR

Reginald N. Meeson

January 1989

Prepared for
STARS Joint Program Office

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 88-033367

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

Approved for public release, unlimited distribution; 30 August 1990. Unclassified.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1989	Final

4. TITLE AND SUBTITLE
Ada Lexical Analyzer Generator

5. FUNDING NUMBERS
MDA 903 84 C 0031
A-134

6. AUTHOR(S)
Reginald N. Meeson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2108

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
STARS Joint Program Office
1400 Wilson Blvd.
Arlington, VA 22209-2308

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution; 30 August 1990.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT *(Maximum 200 words)*

IDA Paper P-2108, Ada Lexical Analyzer Generator, documents the Ada Lexical Analyzer Generator program which will create a lexical analyzer or "next-token" procedure for use in a compiler, pretty printer, or other language processing programs. Lexical analyzers are produced from specifications of the patterns they must recognize. The notation for specifying patterns is essentially the same as that used in the Ada Language Reference Manual. The generator produces an Ada package that includes code to match the specified lexical patterns and returns the symbols it recognizes. Familiarity with compiler terminology and techniques is assumed in the technical sections of this document.

14. SUBJECT TERMS
Ada Programming Language; Software Engineering; Lexical Analysis; Lexical Patterns.

15. NUMBER OF PAGES
174

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

IDA PAPER P-2108

Ada LEXICAL ANALYZER GENERATOR

Reginald N. Meeson

January 1989

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031

DARPA Assignment A-134

PREFACE

The purpose of IDA Paper P-2108, *Ada Lexical Analyzer Generator*, is to forward software and supporting documentation developed as part of IDA's prototype software development work for the Software Technology for Adaptable and Reliable Software (STARS) program under Task Order T-D5-429. This paper documents the Ada Lexical Analyzer Generator's requirements, design, and implementation, and is directed toward potential users who may wish to modify or extend the generator's capabilities.

An earlier draft of this document was reviewed within the Computer and Software Engineering Division (CSED) by B. Brykczynski, W. Easton, R. Knapper, J. Sensiba, L. Veren, R. Waychoff, and R. Winner (April 1988).

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

This document describes the Ada Lexical Analyzer Generator developed as part of IDA's prototype software development work for STARS (Software Technology for Adaptable and Reliable Systems). This report and the accompanying software were written in partial fulfillment of Section 4(d) of Task Order T-D5-429.

The Ada Lexical Analyzer Generator is a program that will create a lexical analyzer or "next-token" procedure for use in a compiler, pretty printer, or other language processing program. Lexical analyzers are produced from specifications of the patterns they must recognize. The notation for specifying patterns is essentially the same as that used in the Ada language reference manual [1]. The generator produces an Ada package that includes code to match the specified lexical patterns and return the symbols it recognizes. Familiarity with compiler terminology and techniques is assumed in the technical sections of this report.

1.1 Scope

This report describes the requirements for the lexical analyzer generator and the approach taken in the prototype design. The report includes descriptions of the notation used for specifying lexical patterns and the internal data structures and processing performed to transform these specifications into Ada pattern recognition code.

1.2 Background

Lexical analysis is the first stage of processing in a compiler or other language processing program, and is where basic language elements such as identifiers, numbers, and special symbols are separated from the sequence of characters submitted as input. Lexical analysis does not include recognizing higher levels of source language structure such as expressions or statements. This processing is performed in the next compiler stage, the parser. Separating the lexical analysis stage from the parsing stage greatly simplifies the parser's task. Lexical analyzers also simplify language processing tools that do not need full-scale parsers to perform their functions; for example, pretty printers. In fact, lexical analysis techniques can simplify many other applications that process complex input data.

For more information on compiler organization and implementation techniques, readers may wish to consult a standard text on compiler development. (See, for example, the "dragon" book [2].)

A lexical analyzer generator produces lexical analyzers automatically from specifications of the input language's lexical components. This is easier and more reliable than coding lexical analyzers manually. One commercial lexical analyzer generator now available is the UNIX®-based program "lex" [3]. The lexical analyzer generator we developed differs from lex in at least three significant ways:

- The notation for describing lexical patterns is much easier to read and understand
- The generator produces directly executable code (lex-generated analyzers are table driven)
- The generator produces Ada code

® UNIX is a registered trademark of AT&T Bell Laboratories.

2. REQUIREMENTS STATEMENT

The Ada Lexical Analyzer Generator is to be a reusable tool for creating lexical analyzers. The generator will translate specifications of lexical patterns into an Ada package or procedure that produces a stream of lexical data values from an input character stream. The notation for specifying lexical patterns should be easy to read and understand. The generator and generated code should be portable. Generated code should be efficient enough for use in practical applications. Documentation is to include descriptions of the development approach, the lexical pattern notation, and the translation techniques employed. A user's guide is also required.

3. DEVELOPMENT APPROACH

The Ada Lexical Analyzer Generator was developed in two stages. The first stage was to define the notation for specifying lexical patterns and actions to be taken when patterns are recognized. The notation we adopted is very similar to that used in the Ada language reference manual to define Ada's syntax. (See [1], Sec. 1.5.)

The second stage of the project was to build the generator, which translates pattern definitions into Ada pattern-matching code. An existing compiler-writing system, Zuse [4], was used to facilitate this work. Using Zuse simplified the parsing of lexical specifications and provided a framework for transforming patterns and generating code.

Figure 1 shows the major components used to build the generator. Starting at the upper left, a translation grammar for the lexical pattern notation was created and processed by the compiler writing system to extract translation action code and a set of run-time translation tables. The translation action code was then compiled together with a skeleton main procedure, support routines, and a bootstrap lexical analyzer to produce the generator.

Figure 1 also shows the second step, which was to generate a new lexical analyzer to replace the bootstrap analyzer. This required creating a set of lexical pattern definitions and running the generator with its run-time translation tables. This use of the generator to produce its own lexical analyzer served as a test of the system. The specification for the replacement lexical analyzer is included in Appendix A.

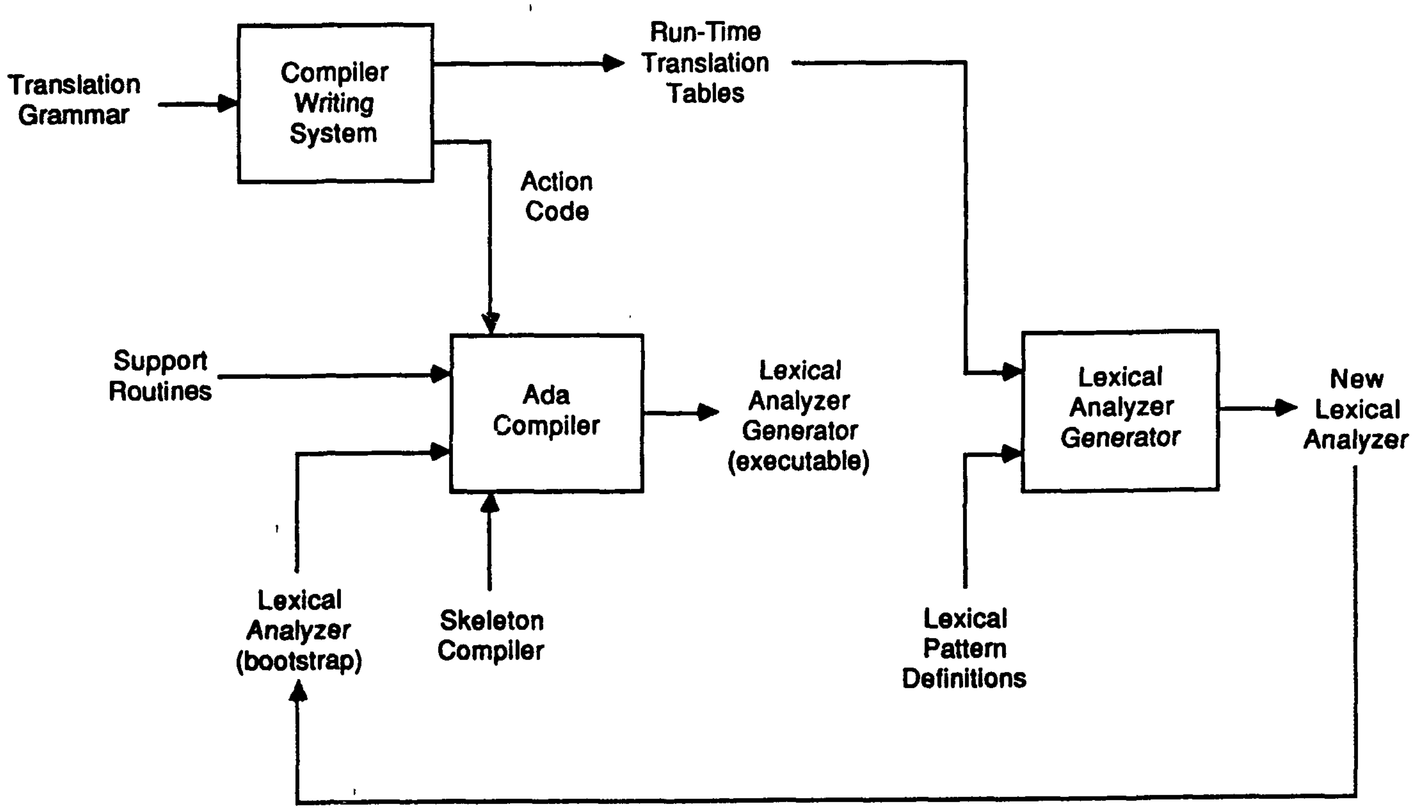

Figure 1. Lexical Analyzer Generator Development Approach

4. DESIGN SPECIFICATION

Designs for translators typically involve two separate designs: one for code that will be generated and one for the translator itself. For this project we also designed the input notation. To explain how these pieces fit together, the design specification is divided into the following five parts:

- The notation used to specify lexical patterns
- Specification of actions to be taken when patterns are recognized
- The input and output interfaces for generated lexical analyzers
- Data structures used to represent patterns within the translator
- Templates used to generate pattern matching code

4.1 Lexical Pattern Notation

Lexical patterns are specified using a simple variant of Backus-Naur Form (BNF). Definitions in this language have the form

```
non_terminal ::= regular_expression ;
```

Non-terminal symbols are represented by Ada identifiers. Regular expressions are made up of terminal and non-terminal symbols using the combining forms described below. Terminal symbols are represented by Ada character and string literals and by reserved identifiers. For example,

```
semicolon ::= ';' ;
apostrophe ::= ''' ;
assignment_symbol ::= ":=" ;
```

Concatenated terms are written consecutively, without an operator symbol, as in

```
character_literal ::= apostrophe graphic_character apostrophe ;
```

Literal string values are equivalent to the concatenation of the corresponding literal characters. For example, the string ":=" is the same as the concatenation of the two characters ':' and '='.

Character ranges can be specified using Ada's double-dot notation. For example,

```
upper_case_letter ::= 'A' .. 'Z' ;
```

A vertical bar is used to separate alternative terms, as in

```
letter_or_digit ::= letter | digit ;
```

Square brackets enclose optional terms. For example, numbers with optional fraction and exponent parts can be specified by

```
numeric_literal ::= integer ['.' integer] [exponent] ;
```

Braces enclose a repeated pattern, as in the following expression for identifiers. The enclosed pattern may be matched zero or more times.

```
identifier ::= letter { ['_'] letter_or_digit } ;
```

Options and repetitions are exercised whenever possible so that the longest possible pattern is matched.

Precedence of operations. Range construction is the highest precedence operation, concatenation is next, and alternation is the lowest.

Look-ahead and ambiguity. Two different patterns may start with the same character or sequence of characters. This situation requires lexical analyzers to look ahead into the input to determine which pattern to match. Look-ahead processing can usually be handled completely automatically.

Patterns may also be ambiguous. That is, a given sequence of characters may match two different patterns at the same time. Normal processing attempts to match the longer pattern first and accept it if it matches. If the longer pattern fails to match, the analyzer will fall back and match the shorter pattern.

To match the shorter of two ambiguous patterns, a special look-ahead operator is provided. The classic situation where this occurs is the Fortran "DO" statement. The following Fortran statements illustrate the problem:

DO 10 I = 1, 10

and

DO 10 I = 1.10

The first is the start of a loop structure, for which the keyword "DO" must be matched. The second is an assignment statement, for which the identifier "DO10I" must be matched. Without special attention, the analyzer would match identifier "DO10I" in both cases. The pattern required to recognize the keyword "DO" is

keyword_DO ::= "DO" # label identifier '=' index_expr ',' ;

The sharp symbol (#, not in quotes) separates this pattern into two parts. If the entire pattern is matched, the analyzer falls back to the # and returns the first part of the pattern as the result. The string to the right is preserved as input to be scanned for the next symbol, which in this example is the loop label. If the pattern fails to match, the lexical analyzer falls back to the # and attempts to match the alternative pattern, which in this example is an identifier.

Regular form. To allow simple, efficient code to be generated for lexical analyzers, the input pattern definitions must have a simple structure. Specifically, they must form a regular grammar so that code for an equivalent finite-state machine can be generated. The pattern construction operations described above allow the definition of arbitrary regular patterns. The lexical analyzer generator does not support recursive pattern definitions.

Predefined patterns. The patterns END_OF_INPUT, END_OF_LINE, and UNRECOGNIZED are automatically defined and handled by the generated code.

Additional examples of pattern definitions can be found in Appendix A.

4.2 Declarations and Actions

In addition to the specification of lexical patterns, the lexical analyzer generator requires definitions of the actions to be taken when a pattern is recognized. These actions may require type, variable, and procedure definitions to be included in the generated code. A lexical analyzer specification, therefore, has the form:

lexicon token_stream_name is

```
        [ declarative_part ]
patterns
        { pattern_definition }
actions
        { action_alternative }
end [token_stream_name] ;
```

"Lexicon" is a reserved word. The token_stream_name is the name of the token stream package generated by the lexical analyzer. The declarative part allows the declaration of any supporting constants, types, variables, functions, or procedures. These declarations are copied into the generated package body.

"Patterns" is a reserved word. Pattern definitions have the form described in Section 4.1.

"Actions" is a reserved word. Action alternatives have the same form as Ada case statement alternatives, i.e.,

```
action_alternative ::=
        when choice {'|' choice} => sequence_of_statements
```

Action choices can be any non-terminal symbol defined in a production or "others" for the last action alternative. The generator turns the action alternatives into a case statement with the name of the recognized pattern as the selector.

There are two principle types of action performed by a lexical analyzer: returning a token value and skipping over uninteresting input. To return a token to the calling program, the action statements must assign a value to the output parameter NEXT (see Section 4.3) and end with a "return" statement. For example,

```
when Identifier =>
        NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM );
        return;
```

To skip over a recognized pattern (for example, white space or comments), specify "null" as the action, with no return. The parameterless function CURRENT_SYMBOL returns the recognized string. CUR_LINE_NUM is an integer variable that holds the current line number.

Examples of action statements are given in Appendix A.

4.3 Input and Output Streams

The input character stream for the lexical analyzer is represented by a procedure that produces consecutive characters each time it is called. The specification for this procedure is:

```
procedure GET_CHARACTER( EOS: out BOOLEAN;
                         NEXT: out CHARACTER;
                         MORE: in BOOLEAN := TRUE );
```

This mechanism allows input text to be produced from a file or from other sources within a program.

The output stream produced by the lexical analyzer generator is a sequence of tokens. The specification for the generated token stream package is:

```
package TOKEN_STREAM_NAME is

    procedure ADVANCE( EOS: out BOOLEAN;
                       NEXT: out TOKEN;
                       MORE: in BOOLEAN := TRUE );

end TOKEN_STREAM_NAME;
```

The package name is taken from the lexicon specification. The procedure ADVANCE reads input by invoking the GET_CHARACTER procedure and returns an end-of-stream flag, EOS, which is TRUE when the end of the input is reached. When EOS is FALSE, NEXT contains the next token value. TOKEN is a user-defined type. The optional parameter MORE may be set to FALSE to indicate that no more tokens will be drawn from the stream.

There are three methods for combining generated stream packages with the remainder of an application program:

- Copying the generated text into the program source file
- Making the generated package body a separate compilation unit
- Creating a generic package

Copying generated text is the least flexible method. If you change any of the lexical patterns, you have to delete the old text and add the new using a text editor. Creating a generic package requires passing the GET_CHARACTER procedure and TOKEN type, and possibly other information, as instantiation parameters. Making the package

body a separate compilation unit is the simplest method. Generics and separate compilation are supported by the generator by allowing either a generic formal part or a "separate" declaration to precede a lexical analyzer specification. A complete description of the form of a specification is:

```
[ context_clause ]
[ generic_formal_part | separate (parent_name) ]
lexicon token_stream_name is
        [ declarative_part ]
patterns
        { production }
actions
        { action_alternative }
end [token_stream_name] ;
```

For generic lexical analyzers, a complete package definition (specification and body) with the specified generic parameters is generated. The GET_CHARACTER procedure and TOKEN type must be included in the list of generic parameters. For non-generic analyzers, only the package body is generated. If a "separate" clause is supplied in the lexicon specification, it is reproduced in the generated code. The parent unit must include the package specification and an "is separate" declaration for the package body.

4.4 Pattern Representation

Pattern definitions are stored as tree structures within the lexical analyzer generator. These tree structures are derived directly from the input specification's parse tree. Pattern trees have nodes corresponding to the following types of patterns:

- Alternation – letter | digit
- Concatenation– letter digit
- Identifier – identifier
- Look-ahead – "DO" # loop_parameters
- Option – [exponent]
- Range – 'A'..'Z'
- Repetition – {digit}

Character literals are represented by range nodes, where the range contains a single character. Empty patterns are represented as empty ranges. Strings are represented as concatenations of individual characters (ranges).

Selection sets. Each pattern has a selection set, which is the set of possible initial

characters. For example, if an identifier starts with an upper- or lower-case letter, then the upper- and lower-case letters will form the pattern's selection set. This information is used to control pattern matching decisions in the generated code. The selection set for a range pattern is the range itself. The selection set for an alternation pattern is the union of the two alternative selection sets. The selection set for a concatenation pattern is that of the left-hand pattern, unless this pattern is an option or repetition, in which case the selection set is the union of the left and right parts.

Ambiguity. Patterns with overlapping selection sets are said to be ambiguous. This is because pattern matching code cannot determine which of the possible alternative patterns to pursue. An example of this is:

dots ::= '.' | "..";

The selection sets for the two alternatives are the same. The ambiguity is that two dots in the input stream could be interpreted either as two consecutive single dots or as one double-dot symbol.

The following transformation eliminates overlapping selection sets and converts ambiguous patterns into unambiguous form. Assume the original pattern is defined by:

```
original ::= left | right ;
left ::= left_overlap | left_unique ;
right ::= right_overlap | right_unique ;
```

where left_overlap and right_overlap have the same selection sets, and the selection sets for left_unique and right_unique are disjoint. (The unique patterns may be empty.) The replacement pattern is:

```
replacement ::= left_unique | new_overlap | right_unique ;
```

where the new overlap pattern is constructed from the overlapping range and the tails of the left and right overlapping patterns:

```
new_overlap ::= overlap_range overlap_tails ;
overlap_tails ::= left_overlap_tail | right_overlap_tail;
```

The tail of a range is the empty pattern, the tail of an alternation is the alternation of the tails of the two components, and the tail of a concatenation is the tail of the left term concatenated with the right.

Resulting patterns can be simplified when any of the component patterns are empty. For example, the result of transforming the dots pattern above is:

```
new_dots ::= '.' ['.'] ;
```

Canonical form. Several additional tree transformations are performed to further simplify pattern tree structures. These are designed to simplify and improve the code that is generated. When a tree satisfies the following conditions it is said to be in canonical form:

- All ambiguities are eliminated
- Alternations of ranges are merged into single range nodes
- Alternations are right associative
- Concatenations are right associative
- Alternations with options or repetitions are treated as option patterns
- Redundant options and repetitions are eliminated

4.5 Code Generation

Package structure. Generated package bodies contain the following major sections:

- Local constant, type, and variable declarations
- Utility function and procedure definitions
- Procedure ADVANCE
- Procedure SCAN_PATTERN

Declarations that appear in lexicon specifications are copied into the first section with the lexical analyzer's local declarations. In addition, the enumerated type PATTERN_ID is created from the list of pattern names defined in the lexicon.

The utility functions and procedures are independent of the lexicon specification and are simply written to the output file.

Action statements are copied from the lexicon specification into the following template for the ADVANCE procedure:

```
procedure ADVANCE ( EOS: out BOOLEAN;
                    NEXT: out TOKEN;
                    MORE: in BOOLEAN := TRUE ) is
```

```
begin
        EOS := FALSE;
        loop
                SCAN_PATTERN;
                case CUR_PATTERN is
                        when END_OF_INPUT =>
                                EOS := TRUE;
                                return;
                        when END_OF_LINE =>
                                null;
                        <lexicon action statements>
                end case;
        end loop;
end ADVANCE;
```

The procedure SCAN_PATTERN contains all of the pattern matching code. The body of this procedure is generated from the canonical-form pattern tree by traversing the tree and filling in standard code templates based on information stored in the tree nodes. The code templates for each pattern type are described below.

Alternation patterns. The template for alternation patterns is:

```
case CURRENT_CHAR is
        <code for each alternative>
        when others =>
                <code for "others" alternative>
end case;
```

The template for each alternative is:

```
when <alternative selection set> =>
        <code to match this alternative>
```

Code to match each alternative is generated by passing the current node's left subtree to the general pattern code generator. Successive alternatives are found by traversing the node's right subtree. If an alternative pattern is part of an option or repetition pattern, the code for the "when others" clause is "null;". Otherwise, if the current character is not included in any of the selection sets, the pattern fails to match.

Concatenation patterns. The template for concatenation patterns when the left subtree could fail is:

```
<code to match left subtree>
if <left subtree didn't fail> then
        <code for right subtree>
end if;
```

If the left subtree cannot fail, the "if" statement is not necessary and this template can be simplified to:

```
<code to match left subtree>
<code for right subtree>
```

Code to match the left subtree is generated by a recursive call to the general pattern code generator. Code for the right subtree is generated the same way if it is an alternation, look-ahead, option, or repetition pattern. When the right subtree is a concatenation or range pattern, the following template is used:

```
case CURRENT_CHAR is
        when <right subtree selection set> =>
                <code to match right subtree>
        when others =>
                <this pattern fails to match>
end case;
```

Option patterns. The template for matching option patterns is:

```
case CURRENT_CHAR is
        when <option selection set> =>
                <code to match the option>
        when others => null;
end case;
```

Code to match the option expression is generated by a recursive call to the general pattern code generator.

Repetition patterns. The template for matching repetition patterns is:

```
loop
        case CURRENT_CHAR is
                <code for the repeated pattern>
                when others => exit;
        end case;
        <exit when look-ahead failed>
end loop;
```

If the repeated pattern is an alternation, "when" clauses for each of the alternatives are generated. Otherwise, a single "when" clause is emitted and code to match the repeated expression is generated by a recursive call to the general pattern code generator. The template for this is:

```
when <repetition selection set> =>
        <code to match repeated pattern>
```

Range patterns. The code generated for range patterns is simply:

```
CHAR_ADVANCE;
```

5. TEST PLAN

The primary test for the lexical analyzer generator was to have it reproduce its own lexical analyzer. This test exercised most of the translator's capabilities. Additional tests were created during development to exercise each type of pattern, check internal tree structures for canonical form, and verify generated code. These test input data and the corresponding results are included in Appendix C.

A test of the generator's portability was conducted by moving the program from its development environment, a DEC® VAX® system with the DEC Ada compiler, to a Sun Workstation® with the Verdix® Ada compiler. The following discrepancies were encountered:

1. Package INTEGER_TEXT_IO, which allows the generator to read and write integer values is predefined in DEC's Ada run-time library. This feature is not standard but it is fully documented. The correction required adding the following two-line package definition and linking it with the program.

```
with TEXT_IO;

INTEGER_TEXT_IO is new INTEGER_IO( INTEGER );
```

2. File STANDARD_ERROR, which the generator uses to report translation errors, is predefined in the Verdix TEXT_IO package. This feature is not standard nor is it described in Verdix's documentation. The correction required deleting the file declaration from package LL_DECLARATIONS and removing the CREATE and CLOSE operations in the main procedure, LL_COMPILE. The entire program then had to be recompiled.

The only other differences between the DEC and Sun environments were in the program-to-file-system interface. A command file was used on the DEC system to connect all input and output files to the program. On the Sun, a link named "TABLE" was

® DEC and VAX are registered trademarks of the Digital Equipment Corporation.

® Sun Workstation is a registered trademark of Sun Microsystems, Inc.

® Verdix is a registered trademark of Verdix Corporation.

created to connect the translation-table file and the UNIX command interpreter was used to connect the standard input and output streams to the source and object code files.

6. REFERENCES

[1] *Ada Programming Language,* ANSI/MIL-STD-1815A, January 1983.

[2] Aho, A., R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, 1985.

[3] Lesk, M., *Lex—A Lexical Analyzer Generator,* Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[4] Pyster, A., *Zuse User's Manual,* Department of Computer Science Technical Report, TRCS81-04, University of California, Santa Barbara, May 1981.

APPENDIX A

Example Lexical Analyzer Specification and Generated Code

This appendix contains two listings. The first is the specification for the lexical analyzer used in the lexical analyzer generator. The second is a pretty-printed version of the code generated by the lexical analyzer generator from the specification. The generator was developed using a hand-written, bootstrap lexical analyzer until it was able to produce this code automatically from the specification.

Contents:

```
separate ( LL_COMPILE )

lexicon  LL_TOKENS  is

   --  This lexicon produces the token stream generator for the
   --  lexical analyzer generator, ADALEX.

patterns

   Graphic_Character  ::=  ' ' .. '~' ;

   Letter  ::=  'A' .. 'Z'  |  'a' .. 'z' ;

   Digit  ::=  '0' .. '9' ;

   Letter_or_Digit  ::=  Letter  |  Digit ;

   Character_Literal  ::=  ''' Graphic_Character ''' ;

   Comment  ::=  "--" { Graphic_Character } ;

   Delimiter  ::=  '&'   |  '''   |  '*'   |  "**"  |  '+'
                |  ','   |  '-'   |  '/'   |  "/="  |  ':'
                |  ":="  |  '<'   |  "<<"  |  "<="  |  "<>"
                |  '='   |  '>'   |  ">="  |  ">>"  ;

   Identifier  ::=  Letter { ['_'] Letter_or_Digit } ;

   Number  ::=  Digit { ['_'] Digit } ;

   Special_Symbol  ::=  '#'    |  '('   |  ')'   |  '.'   |  ".."
                     |  "::="  |  ';'   |  "=>"  |  '['   |  ']'
                     |  '{'    |  '|'   |  '}'   ;

   String_Literal  ::=  Quoted_String { Quoted_String } ;

   Quoted_String  ::=  '"' { Non_Quote_Char } '"' ;
```

```
   Non_Quote_Char  ::=  ' ' ..  '!'  |  '#' .. '~' ;

   White_Space  ::=  Separator { Separator } ;

   Separator  ::=  ' ' | ASCII.HT ;

actions

   when  Character_Literal  =>
      NEXT := MAKE_TOKEN( CHAR, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  Comment | White_Space  =>  null;

   when  Delimiter | Number | Special_Symbol  =>
      NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  Identifier  =>
      NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when  String_Literal  =>
      NEXT := MAKE_TOKEN( STR, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

   when others  =>
      NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM );
      return;

end  LL_TOKENS;
```

```
separate ( LL_COMPILE )

package body LL_TOKENS is

  BUFFER_SIZE: constant := 100;
  subtype BUFFER_INDEX is INTEGER range 1..BUFFER_SIZE;

  type PATTERN_ID is (Character_Literal,Comment,Delimiter,Digit,
                      Graphic_Character,Identifier,Letter,Letter_or_Digit,
                      Non_Quote_Char,Number,Quoted_String,Separator,
                      Special_Symbol,String_Literal,White_Space,
                      END_OF_INPUT, END_OF_LINE, UNRECOGNIZED);

  CUR_LINE_NUM: NATURAL := 0;
  CUR_PATTERN: PATTERN_ID := END_OF_LINE;
  START_OF_LINE: BOOLEAN;
  CHAR_BUFFER: STRING(BUFFER_INDEX);
  CUR_CHAR_NDX: BUFFER_INDEX;
  TOP_CHAR_NDX: BUFFER_INDEX;

  procedure SCAN_PATTERN;  -- forward

  function CURRENT_SYMBOL return STRING is
  begin
    return CHAR_BUFFER(1..(CUR_CHAR_NDX-1));
  end;

  procedure ADVANCE(EOS: out BOOLEAN;
    NEXT: out LLTOKEN;
    MORE: in BOOLEAN := TRUE) is
  begin
    EOS := FALSE;
    loop
      SCAN_PATTERN;
      case CUR_PATTERN is
        when END_OF_INPUT =>
          EOS := TRUE;
          return;
```

```
      when END_OF_LINE => null;
      when Character_Literal =>
        NEXT := MAKE_TOKEN( CHAR, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Comment | White_Space => null;
      when Delimiter | Number | Special_Symbol =>
        NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Identifier =>
        NEXT := MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when String_Literal =>
        NEXT := MAKE_TOKEN( STR, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when others =>
        NEXT := MAKE_TOKEN( LIT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
    end case;
  end loop;
end ADVANCE;


procedure SCAN_PATTERN is

  CURRENT_CHAR: CHARACTER;
  END_OF_INPUT_STREAM: BOOLEAN;
  LOOK_AHEAD_FAILED: BOOLEAN := FALSE;
  FALL_BACK_NDX: BUFFER_INDEX := 1;
  LOOK_AHEAD_NDX: BUFFER_INDEX;

  procedure CHAR_ADVANCE is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    FALL_BACK_NDX := CUR_CHAR_NDX;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
```

```
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

  procedure LOOK_AHEAD is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

begin
  START_OF_LINE := CUR_PATTERN = END_OF_LINE;
  if START_OF_LINE then
    CUR_LINE_NUM := CUR_LINE_NUM+1;
    TOP_CHAR_NDX := 1;
    GET_CHARACTER(END_OF_INPUT_STREAM,CHAR_BUFFER(1));
    if END_OF_INPUT_STREAM then
      CHAR_BUFFER(1) := ASCII.etx;
    end if;
  else
    TOP_CHAR_NDX := TOP_CHAR_NDX-CUR_CHAR_NDX+1;
    for N in 1..TOP_CHAR_NDX loop
      CHAR_BUFFER(N) := CHAR_BUFFER(N+CUR_CHAR_NDX-1);
    end loop;
  end if;
```

```
CUR_CHAR_NDX := 1;
CURRENT_CHAR := CHAR_BUFFER(1);
case CURRENT_CHAR is
  when ASCII.etx =>
    CUR_PATTERN := END_OF_INPUT;
  when ASCII.lf..ASCII.cr =>
    CUR_PATTERN := END_OF_LINE;
  when '"' =>
    CHAR_ADVANCE;
    case CURRENT_CHAR is
      when ' '..'~' =>
        loop
          case CURRENT_CHAR is
            when ' '..'!' | '#'..'~' =>
              CHAR_ADVANCE;
            when others => exit;
          end case;
        end loop;
        case CURRENT_CHAR is
          when '"' =>
            CHAR_ADVANCE;
            CUR_PATTERN := String_Literal;
            loop
              case CURRENT_CHAR is
                when '"' =>
                  LOOK_AHEAD;
                  case CURRENT_CHAR is
                    when ' '..'~' =>
                      loop
                        case CURRENT_CHAR is
                          when ' '..'!' | '#'..'~' =>
                            LOOK_AHEAD;
                          when others => exit;
                        end case;
                      end loop;
                      case CURRENT_CHAR is
                        when '"' =>
                          CHAR_ADVANCE;
```

```
                    when others =>
                      CUR_CHAR_NDX := FALL_BACK_NDX;
                      LOOK_AHEAD_FAILED := TRUE;
                  end case;
                when others =>
                  CUR_CHAR_NDX := FALL_BACK_NDX;
                  LOOK_AHEAD_FAILED := TRUE;
              end case;
            when others => exit;
          end case;
        exit when LOOK_AHEAD_FAILED;
        end loop;
      when others =>
        CUR_PATTERN := UNRECOGNIZED;
    end case;
  when others =>
    CUR_PATTERN := UNRECOGNIZED;
end case;
when 'A'..'Z' | 'a'..'z' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Identifier;
  loop
    case CURRENT_CHAR is
      when '_' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when 'A'..'Z' | 'a'..'z' =>
            CHAR_ADVANCE;
          when '0'..'9' =>
            CHAR_ADVANCE;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when 'A'..'Z' | 'a'..'z' =>
        CHAR_ADVANCE;
      when '0'..'9' =>
        CHAR_ADVANCE;
```

```
      when others => exit;
    end case;
  exit when LOOK_AHEAD_FAILED;
  end loop;
when '0'..'9' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Number;
  loop
    case CURRENT_CHAR is
      when '0'..'9' =>
        CHAR_ADVANCE;
      when '_' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when '0'..'9' =>
            CHAR_ADVANCE;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when others => exit;
    end case;
  exit when LOOK_AHEAD_FAILED;
  end loop;
when ASCII.HT | ' ' =>
  CHAR_ADVANCE;
  CUR_PATTERN := White_Space;
  loop
    case CURRENT_CHAR is
      when ASCII.HT | ' ' =>
        CHAR_ADVANCE;
      when others => exit;
    end case;
  end loop;
when '-' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Delimiter;
  case CURRENT_CHAR is
```

```
    when '-' =>
      CHAR_ADVANCE;
      CUR_PATTERN := Comment;
      loop
        case CURRENT_CHAR is
          when ' '..'~' =>
            CHAR_ADVANCE;
          when others => exit;
        end case;
      end loop;
    when others => null;
  end case;
when ''' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Delimiter;
  case CURRENT_CHAR is
    when ' '..'~' =>
      LOOK_AHEAD;
      case CURRENT_CHAR is
        when ''' =>
          CHAR_ADVANCE;
          CUR_PATTERN := Character_Literal;
        when others =>
          CUR_CHAR_NDX := FALL_BACK_NDX;
          LOOK_AHEAD_FAILED := TRUE;
      end case;
    when others => null;
  end case;
when '&' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Delimiter;
when '*' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Delimiter;
  case CURRENT_CHAR is
    when '*' =>
      CHAR_ADVANCE;
    when others => null;
```

```
    end case;
  when '+'..',' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Delimiter;
  when '/' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Delimiter;
    case CURRENT_CHAR is
      when '=' =>
        CHAR_ADVANCE;
      when others => null;
    end case;
  when ':' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Delimiter;
    case CURRENT_CHAR is
      when '=' =>
        CHAR_ADVANCE;
      when ':' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when '=' =>
            CHAR_ADVANCE;
            CUR_PATTERN := Special_Symbol;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when others => null;
    end case;
  when '<' =>
    CHAR_ADVANCE;
    CUR_PATTERN := Delimiter;
    case CURRENT_CHAR is
      when '<'..'>' =>
        CHAR_ADVANCE;
      when others => null;
    end case;
```

```
      when '=' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Delimiter;
        case CURRENT_CHAR is
          when '>' =>
            CHAR_ADVANCE;
            CUR_PATTERN := Special_Symbol;
          when others => null;
        end case;
      when '>' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Delimiter;
        case CURRENT_CHAR is
          when '='..'>' =>
            CHAR_ADVANCE;
          when others => null;
        end case;
      when '#' | '('..')' | ';' | '[' | ']' | '{'..'}' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Special_Symbol;
      when '.' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Special_Symbol;
        case CURRENT_CHAR is
          when '.' =>
            CHAR_ADVANCE;
          when others => null;
        end case;
      when others =>
        CHAR_ADVANCE;
        CUR_PATTERN := UNRECOGNIZED;
    end case;
  end;

end LL_TOKENS;
```

APPENDIX B

Lexical Analyzer Generator Source Listings

This appendix contains nine listings that make up the lexical analyzer generator. The first is a command file for creating the generator using the Digital Equipment Corporation VAX Ada environment. The second is the translation grammar for the notation used to specify lexical analyzers. The next three are the declarations, action code, and run-time translation tables produced by the translator writing system. The sixth is the skeleton main procedure for the generator. The seventh is the lexical analyzer used to bootstrap the generator. And the last two are the specification and body of the translation support package.

Contents:

```
$ ! make_adalex command file
$ !    usage: @make_adalex
$ !    first process the translation grammar
$ ! @adagen adalex
$ !    then compile the Ada modules in this order:
$ ada ll_decls,ll_compile,ll_tokens,ll_sup_spec,ll_sup_body,ll_actions
$ !    link all the pieces together
$ acs link ll_compile
$ !    and rename the result to "adalex"
$ ren ll_compile.exe adalex.exe
$ exit
```

```
                    (*   Ada Lexical Analayzer Generator    *)


(*  This grammar defines the input notation used to specify the lexical
    elements of a language in the Ada Lexical Analyzer Generator [1].
    This notation is essentially the same as the BNF notation used in
    the Ada language reference manual (ANSI/MIL-STD-1815A).  The trans-
    lation of a collection of lexical pattern definitions and associated
    actions yields an Ada package body that contains a "next-token"
    procedure for use in a compiler or other language processing tool.  *)

(*  This grammar is an LL(1) grammar that is processed by an Ada version
    of the Zuse parser generator [2].  Zuse produces a complete table-
    driven translator that recognizes the specified productions and
    applies the indicated translation actions.  *)


(*  Author:  Reg Meeson, grammar 7/31/87, actions 12/15/87  *)


(*  References:  *)

(*  1. Meeson, R., The Ada Lexical Analyzer Generator, Institute
       for Defense Analyses, Paper P-3008, May 1988.  *)

(*  2. Pyster, A., Zuse User's Manual, University of California,
       Santa Barbara, Department of Computer Science, Technical
       Report TRCS81-04, May 1981.  *)


%a  (* Axiom: *)

    Adalex

%n  (* Non-terminal symbols: *)

    ActionCode     ActionDefs     AltChoices     Alternation    Catenation
    Character      CharOrRange    Declarations   DeclStmt       DeclToken
```

```
    IdentOrOther   LookAhead      NonEndCode     NonEndOrWhen   NonEndStmt
    NonWhenStmt    NonWhenToks    OptIdent       PatternDefs    RangePart
    RegularExpr    RegularFact    RegularTerm    SubunitDecl

%g  (* Terminal symbol groups: *)

    CharLit        Identifier     Other          StringLit

%l  (* Literal terminal symbols: *)

    #              (              )              .              ..
    ::=            ;              =>             ASCII          [
    ]              actions        begin          case           end
    if             is             lexicon        loop           others
    patterns       separate       when           {              |
    }

%t  (* Declarations for the generated translator: *)

    ANONYMOUS: constant LLSTRINGS := "ANONYMOUS           ";

    type  NODE_TYPE  is  ( ALT, BAD, CAT, CHAR, EMPTY, IDENT,
                           LIT, LOOK, OPT, REP, RNG, STR );

    type  SELECTION_SET  is  array ( ASCII.HT .. '~' ) of BOOLEAN;

    type  TREE_NODE ( VARIANT: NODE_TYPE );

    type  LLATTRIBUTE  is access  TREE_NODE;

    type  TREE_NODE ( VARIANT: NODE_TYPE )  is
      record
        NAME: LLSTRINGS;
        SEL_SET: SELECTION_SET;
        NULLABLE: BOOLEAN;
        COULD_FAIL: BOOLEAN;
        case  VARIANT  is
          when  ALT | CAT | LOOK  =>  LEFT, RIGHT: LLATTRIBUTE;
```

```
            when  BAD | EMPTY | RNG  =>  null;
            when  CHAR  =>  CHAR_VAL: CHARACTER;
            when  IDENT | LIT | STR  =>  STRING_VAL: LLSTRINGS;
            when  OPT | REP  =>  EXPR: LLATTRIBUTE;
          end case;
        end record;

    BAD_PATTERN: constant LLATTRIBUTE :=
        new TREE_NODE'(BAD,ANONYMOUS,(others => FALSE),FALSE,TRUE);

%%  (* Productions: *)

Adalex          =         {s lexicon => 2, patterns => 6, actions => 8, @ => * }
                     SubunitDecl
                     lexicon Identifier is
                          {a PUT("package body"); EMIT_TOKEN($3);
                             PUT_LINE(" is"); }
                        Declarations
                     patterns
                        PatternDefs
                          {a COMPLETE_PATTERNS;
                             EMIT_PKG_DECLS; }
                     actions
                          {a EMIT_ADVANCE_HDR; }
                        ActionDefs
                     end OptIdent ';'
                          {a EMIT_ADVANCE_TLR;
                             EMIT_SCAN_PROC;
                             PUT("end"); EMIT_TOKEN($3); PUT_LINE(";"); } ;

SubunitDecl     =    separate ( Identifier )
                          {a PUT("separate ("); EMIT_TOKEN($3);
                             PUT_LINE(" )"); NEW_LINE; } ;
                =    % any ;

Declarations    =    ;
                =         {s ';' => 2, patterns => *, actions => *, @ => * }
                     DeclStmt ';'
```

```
                        {a EMIT_TOKEN($2); }
                     Declarations  % any ;

DeclStmt        =  DeclToken
                        {a EMIT_TOKEN($1); }
                     DeclStmt;
                =  ;

DeclToken       =  case
                        {a $0 := $1; } ;
                =  end
                        {a $0 := $1; } ;
                =  when
                        {a $0 := $1; } ;
                =  NonEndOrWhen
                        {a $0 := $1; } ;

NonEndOrWhen    =  Character
                        {a $0 := $1; } ;
                =  Identifier
                        {a $0 := $1; } ;
                =  Other
                        {a $0 := $1; } ;
                =  StringLit
                        {a $0 := $1; } ;
                =  is
                        {a $0 := $1; } ;
                =  others
                        {a $0 := $1; } ;
                =  '('
                        {a $0 := $1; } ;
                =  ')'
                        {a $0 := $1; } ;
                =  '.'
                        {a $0 := $1; } ;
                =  '..'
                        {a $0 := $1; } ;
                =  '=>'
```

```
                  {a $0 := $1; } ;
            =  '|'
                  {a $0 := $1; } ;

Character   =  CharLit
                  {a $0 := $1; } ;
            =  ASCII '.' Identifier
                  {a $0 := CVT_ASCII($3); } ;

PatternDefs =  ;
            =     {s Identifier => 1, ::= => 2, actions => *, @ => * }
               Identifier ::=
                  {p LLSKIPNODE; }
                 RegularExpr LookAhead ';'
                  {a STORE_PATTERN($1,CONCATENATE($3,$4)); }
                 PatternDefs  % any ;

RegularExpr =  RegularTerm Alternation
                  {a $0 := ALTERNATE($1,$2); } ;

RegularTerm =  RegularFact
                  {s ';' => *, actions => *, @ => *;
                    $0 := BAD_PATTERN; }
                 Catenation
                  {a $0 := CONCATENATE($1,$2); }  % any ;

RegularFact =  CharOrRange
                  {a $0 := $1; } ;
            =  Identifier
                  {a $0 := $1; } ;
            =  StringLit
                  {a $0 := CVT_STRING($1); } ;
            =  '[' RegularExpr ']'
                  {p LLSKIPNODE; }
                  {a $0 := OPTION($2); } ;
            =  '{' RegularExpr '}'
                  {p LLSKIPNODE; }
                  {a $0 := REPEAT($2); } ;
```

```
CharOrRange    =  Character RangePart
                        {a $0 := CHAR_RANGE($1,$2); } ;

RangePart      =  '..' Character
                        {a $0 := $2; } ;
               =        {a $0 := null; } ;

Catenation     =  RegularFact Catenation
                        {a $0 := CONCATENATE($1,$2); } ;
               =        {a $0 := null; } ;

Alternation    =  '|' RegularExpr
                        {a $0 := $2; } ;
               =        {a $0 := null; } ;

LookAhead      =  '#' RegularExpr
                        {a $0 := LOOK_AHEAD($2); } ;
               =        {a $0 := null; } ;

ActionDefs     =  ;
               =        {s when => 1, @ => * }
                  when
                        {a EMIT_TOKEN($1); }
                     IdentOrOther '=>'
                        {p LLSKIPNODE; }
                        {a EMIT_TOKEN($3); }
                     ActionCode ActionDefs  % any ;

IdentOrOther   =  Identifier
                        {a EMIT_TOKEN($1); INCLUDE_PATTERN($1); }
                     AltChoices ;
               =  others
                        {a EMIT_TOKEN($1); } ;

AltChoices     =  '|' Identifier
                        {a EMIT_TOKEN($1); EMIT_TOKEN($2);
                           INCLUDE_PATTERN($2); }
                     AltChoices ;
```

```
                =  ;

ActionCode      =  ;
                =        {s ';' => 2, when => *, @ => * }
                   NonWhenStmt ';'
                         {a EMIT_TOKEN($2); }
                      ActionCode  % any ;

NonWhenStmt     =  begin
                         {a EMIT_TOKEN($1); }
                      NonEndCode end
                         {a EMIT_TOKEN($3); } ;
                =  case
                         {a EMIT_TOKEN($1); }
                      NonEndCode end case
                         {a EMIT_TOKEN($3); EMIT_TOKEN($4); } ;
                =  if
                         {a EMIT_TOKEN($1); }
                      NonEndCode end if
                         {a EMIT_TOKEN($3); EMIT_TOKEN($4); } ;
                =  loop
                         {a EMIT_TOKEN($1); }
                      NonEndCode end loop
                         {a EMIT_TOKEN($3); EMIT_TOKEN($4); } ;
                =  NonWhenToks ;

NonEndCode      =  NonEndStmt ';'
                         {a EMIT_TOKEN($2); }
                      NonEndCode ;
                =  ;

NonEndStmt      =  NonWhenStmt ;
                =  when
                         {a EMIT_TOKEN($1); }
                      NonWhenToks ;

NonWhenToks     =  NonEndOrWhen
                         {a EMIT_TOKEN($1); }
```

```
                   NonWhenToks ;
              =  ;

OptIdent      =  Identifier
                       {a $0 := $1; } ;
              =        {a $0 := null; } ;

%%
```

```
with TEXT_IO;

package  LL_DECLARATIONS  is

   LLSTRINGLENGTH: constant := 20;

   subtype LLSTRINGS is STRING(1..LLSTRINGLENGTH);

   ANONYMOUS: constant LLSTRINGS := "ANONYMOUS           ";

   type  NODE_TYPE  is  ( ALT, BAD, CAT, CHAR, EMPTY, IDENT,
                          LIT, LOOK, OPT, REP, RNG, STR );

   type  SELECTION_SET  is  array ( ASCII.HT .. '~' ) of BOOLEAN;

   type  TREE_NODE ( VARIANT: NODE_TYPE );

   type  LLATTRIBUTE  is access  TREE_NODE;

   type  TREE_NODE ( VARIANT: NODE_TYPE )  is
      record
         NAME: LLSTRINGS;
         SEL_SET: SELECTION_SET;
         NULLABLE: BOOLEAN;
         COULD_FAIL: BOOLEAN;
         case  VARIANT  is
            when  ALT | CAT | LOOK  =>  LEFT, RIGHT: LLATTRIBUTE;
            when  BAD | EMPTY | RNG  =>  null;
            when  CHAR  =>  CHAR_VAL: CHARACTER;
            when  IDENT | LIT | STR  =>  STRING_VAL: LLSTRINGS;
            when  OPT | REP  =>  EXPR: LLATTRIBUTE;
         end case;
      end record;

   BAD_PATTERN: constant LLATTRIBUTE :=
      new TREE_NODE'(BAD,ANONYMOUS,(others => FALSE),FALSE,TRUE);

   LLTABLESIZE: constant := 32;
```

```
   LLRHSSIZE: constant := 174;
   LLPRODSIZE: constant := 64;
   LLSYNCHSIZE: constant := 26;

   STANDARD_ERROR: TEXT_IO.FILE_TYPE;

end  LL_DECLARATIONS;
```

```
with  LL_SUPPORT;
separate  ( LL_COMPILE )
procedure  LLTAKEACTION( CASEINDEX: in INTEGER )  is
   use  LL_SUPPORT;
begin
   case  CASEINDEX  is
      when 0 => null;
      when 1 =>
         PUT("package body");
         EMIT_TOKEN(llstack(llsentptr-2).attribute);
         PUT_LINE(" is");
      when 2 =>
         COMPLETE_PATTERNS;
         EMIT_PKG_DECLS;
      when 3 =>
         EMIT_ADVANCE_HDR;
      when 4 =>
         EMIT_ADVANCE_TLR;
         EMIT_SCAN_PROC;
         PUT("end");
         EMIT_TOKEN(llstack(llsentptr-13).attribute);
         PUT_LINE(";");
      when 5 =>
         PUT("separate (");
         EMIT_TOKEN(llstack(llsentptr-2).attribute);
         PUT_LINE(" )");
         NEW_LINE;
      when 6 =>
         EMIT_TOKEN(llstack(llsentptr-1).attribute);
      when 7 =>
         EMIT_TOKEN(llstack(llsentptr-1).attribute);
      when 8 =>
         llstack(llstack(llsentptr).parent).attribute :=
            llstack(llsentptr-1).attribute;
      when 9 =>
         llstack(llstack(llsentptr).parent).attribute :=
            llstack(llsentptr-1).attribute;
      when 10 =>
```

```
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 11 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 12 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 13 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 14 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 15 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 16 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 17 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 18 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 19 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 20 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 21 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 22 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
```

```
when 23 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 24 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 25 =>
   llstack(llstack(llsentptr).parent).attribute :=
      CVT_ASCII(llstack(llsentptr-1).attribute);
when 26 =>
   LLSKIPNODE;
when 27 =>
   STORE_PATTERN(llstack(llsentptr-6).attribute,
      CONCATENATE(llstack(llsentptr-3).attribute,
         llstack(llsentptr-2).attribute));
when 28 =>
   llstack(llstack(llsentptr).parent).attribute :=
      ALTERNATE(llstack(llsentptr-2).attribute,
         llstack(llsentptr-1).attribute);
when 29 =>
   llstack(llstack(llsentptr).parent).attribute := BAD_PATTERN;
when 30 =>
   llstack(llstack(llsentptr).parent).attribute :=
      CONCATENATE(llstack(llsentptr-2).attribute,
         llstack(llsentptr-1).attribute);
when 31 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 32 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 33 =>
   llstack(llstack(llsentptr).parent).attribute :=
      CVT_STRING(llstack(llsentptr-1).attribute);
when 34 =>
   LLSKIPNODE;
when 35 =>
   llstack(llstack(llsentptr).parent).attribute :=
```

```
      OPTION(llstack(llsentptr-3).attribute);
when 36 =>
   LLSKIPNODE;
when 37 =>
   llstack(llstack(llsentptr).parent).attribute :=
      REPEAT(llstack(llsentptr-3).attribute);
when 38 =>
   llstack(llstack(llsentptr).parent).attribute :=
      CHAR_RANGE(llstack(llsentptr-2).attribute,
         llstack(llsentptr-1).attribute);
when 39 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 40 =>
   llstack(llstack(llsentptr).parent).attribute := null;
when 41 =>
   llstack(llstack(llsentptr).parent).attribute :=
      CONCATENATE(llstack(llsentptr-2).attribute,
         llstack(llsentptr-1).attribute);
when 42 =>
   llstack(llstack(llsentptr).parent).attribute := null;
when 43 =>
   llstack(llstack(llsentptr).parent).attribute :=
      llstack(llsentptr-1).attribute;
when 44 =>
   llstack(llstack(llsentptr).parent).attribute := null;
when 45 =>
   llstack(llstack(llsentptr).parent).attribute :=
      LOOK_AHEAD(llstack(llsentptr-1).attribute);
when 46 =>
   llstack(llstack(llsentptr).parent).attribute := null;
when 47 =>
   EMIT_TOKEN(llstack(llsentptr-1).attribute);
when 48 =>
   LLSKIPNODE;
when 49 =>
   EMIT_TOKEN(llstack(llsentptr-2).attribute);
when 50 =>
```

```
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
      INCLUDE_PATTERN(llstack(llsentptr-1).attribute);
   when 51 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 52 =>
      EMIT_TOKEN(llstack(llsentptr-2).attribute);
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
      INCLUDE_PATTERN(llstack(llsentptr-1).attribute);
   when 53 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 54 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 55 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 56 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 57 =>
      EMIT_TOKEN(llstack(llsentptr-2).attribute);
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 58 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 59 =>
      EMIT_TOKEN(llstack(llsentptr-2).attribute);
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 60 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 61 =>
      EMIT_TOKEN(llstack(llsentptr-2).attribute);
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 62 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 63 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 64 =>
      EMIT_TOKEN(llstack(llsentptr-1).attribute);
   when 65 =>
      llstack(llstack(llsentptr).parent).attribute :=
         llstack(llsentptr-1).attribute;
```

```
      when 66 =>
         llstack(llstack(llsentptr).parent).attribute := null;
      when others =>
         PUT_LINE( STANDARD_ERROR, "*** Zuse -- Error in action code ***" );
   end case;
end  LLTAKEACTION;
```

```
#              l
(              l
)              l
::=            l
;              l
=>             l
@              g
ASCII          l
CharLit        g
Identifier     g
Other          g
StringLit      g
[              l
]              l
actions        l
any            g
begin          l
case           l
end            l
if             l
is             l
lexicon        l
loop           l
others         l
patterns       l
separate       l
when           l
{              l
|              l
}              l
   1
   1  16   2
n   2   0   1
l  24   0   1
g  12   0   1
l  23   0   1
a   1
n   4   0   1
```

```
l 27 0 1
n 26 0 1
a 2
l 17 0 1
a 3
n 44 0 1
l 21 0 1
n 63 0 1
l 7 0 1
a 4
24 28
2 5 1
l 28
l 2
g 12
l 3
a 5
28
2 0 2
18 24
4 0 1
27
4 4 18
n 6 0 6
l 7 0 6
a 6
n 4 0 6
2 3 4 5 7 8 10 11 12 13 14 18 20 21 23 26 29 31
6 3 16
n 8
a 7
n 6
2 3 4 5 8 10 11 12 13 14 20 21 23 26 29 31
6 0 1
7
8 2 1
l 20
a 8
```

```
  20
   8   2   1
l  21
a   9
  21
   8   2   1
l  29
a  10
  29
   8   2  13
n  12
a  11
   2   3   4   5   8  10  11  12  13  14  23  26  31
  12   2   2
n  24
a  12
  10  11
  12   2   1
g  12
a  13
  12
  12   2   1
g  13
a  14
  13
  12   2   1
g  14
a  15
  14
  12   2   1
l  23
a  16
  23
  12   2   1
l  26
a  17
  26
  12   2   1
```

```
l    2
a   18
     2
    12    2    1
l    3
a   19
     3
    12    2    1
l    4
a   20
     4
    12    2    1
l    5
a   21
     5
    12    2    1
l    8
a   22
     8
    12    2    1
l   31
a   23
    31
    24    2    1
g   11
a   24
    11
    24    4    1
l   10
l    4
g   12
a   25
    10
    26    0    1
    17
    26    8    2
g   12     0   11
l    6     0   11
```

```
p  26
n  28   0  11
n  42   0  11
l   7   0  11
a  27
n  26   0  11
  12  18
  28   3   6
n  29
n  40
a  28
  10  11  12  14  15  30
  29   3   7
n  30  29  16
n  38
a  30
  10  11  12  14  15  18  30
  30   2   2
n  35
a  31
  10  11
  30   2   1
g  12
a  32
  12
  30   2   1
g  14
a  33
  14
  30   5   1
l  15
n  28
l  16
p  34
a  35
  15
  30   5   1
l  30
```

```
n  28
l  32
p  36
a  37
  30
  35   3   2
n  24
n  36
a  38
  10  11
  36   3   1
l   5
n  24
a  39
   5
  36   1  11
a  40
   1   7  10  11  12  14  15  16  30  31  32
  38   3   6
n  30
n  38
a  41
  10  11  12  14  15  30
  38   1   5
a  42
   1   7  16  31  32
  40   3   1
l  31
n  28
a  43
  31
  40   1   4
a  44
   1   7  16  32
  42   3   1
l   1
n  28
a  45
```

```
   1
  42   1   1
a  46
   7
  44   0   1
  21
  44   8   2
l  29   0  20
a  47
n  46   0  20
l   8   0  20
p  48
a  49
n  50   0  20
n  44   0  20
  18  29
  46   3   1
g  12
a  50
n  48
  12
  46   2   1
l  26
a  51
  26
  48   4   1
l  31
g  12
a  52
n  48
  31
  48   0   1
   8
  50   0   2
  21  29
  50   4  19
n  52   0  23
l   7   0  23
```

```
a  53
n  50   0  23
   2   3   4   5   7   8  10  11  12  13  14  18  19  20  22  23  25  26  31
  52   5   1
l  19
a  54
n  57
l  21
a  55
  19
  52   6   1
l  20
a  56
n  57
l  21
l  20
a  57
  20
  52   6   1
l  22
a  58
n  57
l  21
l  22
a  59
  22
  52   6   1
l  25
a  60
n  57
l  21
l  25
a  61
  25
  52   1  14
n  61
   2   3   4   5   7   8  10  11  12  13  14  23  26  31
  57   4  19
```

```
n  59
l   7
a  62
n  57
    2   3   4   5   7   8  10  11  12  13  14  19  20  22  23  25  26  29  31
   57   0   1
   21
   59   1  18
n  52
    2   3   4   5   7   8  10  11  12  13  14  19  20  22  23  25  26  31
   59   3   1
l  29
a  63
n  61
   29
   61   3  13
n  12
a  64
n  61
    2   3   4   5   8  10  11  12  13  14  23  26  31
   61   0   1
    7
   63   2   1
g  12
a  65
   12
   63   1   1
a  66
    7
   24   2
   27   6
   17   8
    9  -1
    0   0
    7   2
   27  -1
   17  -1
    9  -1
```

0	0
12	1
6	2
17	-1
9	-1
0	0
7	-1
17	-1
9	-1
0	0
29	1
9	-1
0	0
7	2
29	-1
9	-1
0	0

```
with  LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;


procedure  LL_COMPILE  is


--      Skeletal compiler to parse a candidate string

--      May 8, 1981
--      Version:  1.0
--      Author:  Arthur Pyster
--      The original Pascal version of this program was copyrighted
--      by The Regents of the University of California

--      August 1984
--      Modified for use on an IBM/PC with IBM's Pascal compiler
--      Change Author:  Reg Meeson

--      November 1986
--      Ported to AT&T UNIX PC7300
--      Change Author:  Reg Meeson

--      August 1987
--      Ported to VAX 8600 and converted from Pascal to Ada
--      Change Author:  Reg Meeson
--      The Ada version of this program was produced for the DoD
--      STARS Program


--      Purpose:  This program is a skeletal compiler which is fleshed
--      out by the inclusion of two packages supplied by the user:

--         LL_SUPPORT -- support routines called directly or indirectly
--            as translation action routines

--         LL_TOKENS -- lexical analysis routines that produce a stream
--            of tokens
```

```
--     and 2 units produced by GENERATE, the parser generator:

--        LL_DECLARATIONS -- constant, type, and variable declarations
--           specified in grammar

--        LLTAKEACTION -- procedure which calls action routines as
--           dictated by grammar rules

--     First, the translation table file produced by GENERATE is read.
--     It contains an encoded form of the literals symbol table, the
--     parsing action tables, and error recovery data.  Source code is
--     read from the STANDARD_INPUT file and object code is written to
--     the STANDARD_OUTPUT file.  Error messages are written to a file
--     called STANDARD_ERROR.


use  LL_DECLARATIONS, INTEGER_TEXT_IO, TEXT_IO;


PARSING_ERROR: exception;    --  for fatal parsing errors


LLMAXSTACK: constant := 500;
   -- max number of sentential form elements in parse tree at one time


type  LLTOKEN  is     -- for tokens produced by the lexical analyzer
   record
      PRINTVALUE: LLSTRINGS;     -- the literal token value
      TABLEINDEX: INTEGER;       -- where token is in symbol table
      LINENUMBER: INTEGER;       -- where token appeared in source
      ATTRIBUTE:  LLATTRIBUTE;  -- user's token attributes
   end record;

type  LLSTYLE  is  (LITERAL, NONTERMINAL, GROUP, ACTION, PATCH);
   -- literal: a terminal that stands for itself
   -- group: a terminal that is a member of a lexical group
   -- action: an action code segment
```

```
   -- patch: action code to patch a syntax error

type  LLSYMTABENTRY  is     -- for symbol table entries
   record
      KEY: LLSTRINGS;    -- literal string or group identifier
      KIND: LLSTYLE;     -- literal or group
   end record;

type  LLRIGHT  is     -- for grammar vocabulary symbols
   record
      CASEINDEX: INTEGER;     -- action code case index
      SYNCHINDEX: INTEGER;    -- synchronization table index
      WHICHCHILD: INTEGER;    -- position in production right hand side
      KIND: LLSTYLE;          -- type of vocabulary symbol
      TABLEINDEX: INTEGER;    -- symbol table or production start index
   end record;

type  LLSENTENTIAL  is     -- for sentential forms
   record
      LASTCHILD: BOOLEAN;       -- is this the rightmost child?
      TOP: INTEGER;             -- pointer to lastchild
      PARENT: INTEGER;          -- pointer to parent of this node
      ATTRIBUTE: LLATTRIBUTE;   -- derived attributes returned
      DATA: LLRIGHT;            -- vocabulary symbol information
   end record;


LLADVANCE: BOOLEAN;     -- advance llsentptr to next node?
LLEOTOKS: BOOLEAN;      -- end of token stream encountered
LLLOCEOS: INTEGER;      -- location of end-of-input in symboltable
LLSENTPTR: INTEGER;     -- current sentential form element
LLCURTOK: LLTOKEN;      -- the current token
LLSYMBOLTABLE: array ( 1 .. LLTABLESIZE ) of LLSYMTABENTRY;
                        -- the symbol table for literal terms
LLSTACK: array ( 1 .. LLMAXSTACK ) of LLSENTENTIAL;
                        -- stack which represents the parse tree
```

```
procedure  LLNEXTTOKEN;
   -- get the next token from the input stream (defined below)


function  LLFIND( ITEM: LLSTRINGS; WHICH: LLSTYLE )  return  INTEGER  is
   --  Find item in symbol table -- return index or 0 if not found.
   --  Assumes symbol table is sorted in ascending order.
   LOW, MIDPOINT, HIGH: INTEGER;
begin
   LOW := 1;
   HIGH := LLTABLESIZE + 1;
   while  LOW /= HIGH  loop
      MIDPOINT := (HIGH + LOW) / 2;
      if  ITEM < LLSYMBOLTABLE(MIDPOINT).KEY  then
         HIGH := MIDPOINT;
      elsif  ITEM = LLSYMBOLTABLE(MIDPOINT).KEY  then
         if  LLSYMBOLTABLE(MIDPOINT).KIND = WHICH  then
            return( MIDPOINT );
         else
            return( 0 );
         end if;
      else   --  ITEM > LLSYMBOLTABLE(MIDPOINT).KEY
         LOW := MIDPOINT + 1;
      end if;
   end loop;
   return( 0 );  -- item is not in table
end  LLFIND;


procedure  LLPRTSTRING( STR: LLSTRINGS )  is
   --  print non-blank prefix of str in quotes
begin
   PUT( STANDARD_ERROR, '"' );
   for  I  in  STR'RANGE  loop
   exit when  STR(I) = ' ';
      PUT( STANDARD_ERROR, STR(I) );
   end loop;
   PUT( STANDARD_ERROR, '"' );
```

```
end  LLPRTSTRING;


procedure  LLPRTTOKEN  is
   --  print the current token
begin
   if  LLCURTOK.PRINTVALUE(1) in '!'..'~'  then   -- printable ASCII
      LLPRTSTRING(LLCURTOK.PRINTVALUE);
   else
      PUT( STANDARD_ERROR,
           "unprintable token beginning with CHARACTER'POS = " );
      PUT( STANDARD_ERROR, CHARACTER'POS(LLCURTOK.PRINTVALUE(1)), 1 );
   end if;
end  LLPRTTOKEN;


procedure  LLSKIPTOKEN  is
   -- remove current token
begin
   LLADVANCE := FALSE;
   PUT( STANDARD_ERROR, "*** Skipping ");
   LLPRTTOKEN;
   PUT_LINE( STANDARD_ERROR, " in line ");
   PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
   PUT_LINE( STANDARD_ERROR, "." );
   L..EXTTOKEN;
end  LLSKIPTOKEN;


procedure  LLSKIPNODE  is
   --  skip over sentential form node leaving current token as is
begin
   PUT( STANDARD_ERROR, "*** Inserting " );
   LLPRTSTRING( LLSYMBOLTABLE(LLSTACK(LLSENTPTR).DATA.TABLEINDEX).KEY );
   PUT( STANDARD_ERROR, " before " );
   LLPRTTOKEN;
   PUT( STANDARD_ERROR, " in line " );
   PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
```

```
   PUT_LINE( STANDARD_ERROR, "." );
   LLSENTPTR := LLSENTPTR + 1;
end  LLSKIPNODE;


procedure  LLSKIPBOTH  is
   --  Skip over both sentential form node and current token.  Used
   --  when replacement is assumed to be correct, and attribute of
   --  replacement does not need to be set; otherwise, use llreplace.
begin
   PUT( STANDARD_ERROR, "*** " );
   LLPRTTOKEN;
   PUT( STANDARD_ERROR, " replaced by " );
   LLPRTSTRING( LLSYMBOLTABLE(LLSTACK(LLSENTPTR).DATA.TABLEINDEX).KEY );
   PUT( STANDARD_ERROR, " in line " );
   PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
   PUT_LINE( STANDARD_ERROR, "." );
   LLSENTPTR := LLSENTPTR + 1;
   LLNEXTTOKEN;
end  LLSKIPBOTH;


procedure  LLFATAL  is
   --  To recover from syntactic error, terminate compilation
begin
   PUT( STANDARD_ERROR, "*** Fatal " );
   LLPRTTOKEN;
   PUT( STANDARD_ERROR, " found in line ");
   PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
   PUT_LINE( STANDARD_ERROR, " -- terminating translation." );
   raise  PARSING_ERROR;
end  LLFATAL;


procedure GET_CHARACTER( EOS: out BOOLEAN;
                         NEXT: out CHARACTER;
                         MORE: in BOOLEAN := TRUE ) is
   --  Produce input characters for the lexical analyzer.
```

```
begin
   if  END_OF_FILE(STANDARD_INPUT)  then
      EOS := TRUE;
   elsif  END_OF_LINE(STANDARD_INPUT)  then
      SKIP_LINE(STANDARD_INPUT);
      EOS := FALSE;
      NEXT := ASCII.CR;
   else
      EOS := FALSE;
      GET(STANDARD_INPUT,NEXT);
   end if;
end;


function  MAKE_TOKEN( NODE: NODE_TYPE; SYMBOL: STRING; LINENUMBER: NATURAL )
      return  LLTOKEN  is
   --  construct a token value from input lexical information
   PRINTVALUE: LLSTRINGS;
   TABLEINDEX: INTEGER;
   ATTRIBUTE: LLATTRIBUTE;

   function CVT_STRING( STR: in STRING ) return LLSTRINGS is
      --  Convert an arbitrary-length string to a fixed length string.
      RESULT: LLSTRINGS;
   begin
      for  I in LLSTRINGS'RANGE  loop
         if  I <= STR'LAST  then
            RESULT(I) := STR(I);
         else
            RESULT(I) := ' ';
         end if;
      end loop;
      return RESULT;
   end;

begin
   PRINTVALUE := CVT_STRING(SYMBOL);
   case  NODE  is
```

```
      when  CHAR  =>
         TABLEINDEX := LLFIND("CharLit              ", GROUP);
      when  IDENT  =>
         TABLEINDEX := LLFIND(PRINTVALUE, LITERAL);
         if  TABLEINDEX = 0  then
            TABLEINDEX := LLFIND("Identifier          ", GROUP);
         end if;
      when  LIT  =>
         TABLEINDEX := LLFIND(PRINTVALUE, LITERAL);
         if  TABLEINDEX = 0  then
            TABLEINDEX := LLFIND("Other               ", GROUP);
         end if;
      when  STR  =>
         TABLEINDEX := LLFIND("StringLit            ", GROUP);
      when others  =>
         TABLEINDEX := 0;
   end case;
   case  NODE  is
      when  CHAR  =>
         ATTRIBUTE := new TREE_NODE'(CHAR,PRINTVALUE,(others => FALSE),
                                    FALSE,FALSE,PRINTVALUE(2));
      when  IDENT  =>
         ATTRIBUTE := new TREE_NODE'(IDENT,PRINTVALUE,(others => FALSE),
                                    FALSE,FALSE,PRINTVALUE);
      when  LIT  =>
         ATTRIBUTE := new TREE_NODE'(LIT,PRINTVALUE,(others => FALSE),
                                    FALSE,FALSE,PRINTVALUE);
      when  STR  =>
         ATTRIBUTE := new TREE_NODE'(STR,PRINTVALUE,(others => FALSE),
                                    FALSE,FALSE,PRINTVALUE);
      when others  =>
         ATTRIBUTE := new TREE_NODE'(BAD,PRINTVALUE,(others => FALSE),
                                    FALSE,FALSE);
   end case;
   return  LLTOKEN'(PRINTVALUE,TABLEINDEX,LINENUMBER,ATTRIBUTE);
end  MAKE_TOKEN;
```

```
package  LL_TOKENS  is
   -- produces a stream of tokens from the STANDARD_INPUT file
   procedure  ADVANCE( EOS: out BOOLEAN;
                       NEXT: out LLTOKEN;
                       MORE: in  BOOLEAN := TRUE );
end  LL_TOKENS;


package body  LL_TOKENS  is separate;


procedure  LLNEXTTOKEN  is
   -- get the next token from the input stream
begin
   LL_TOKENS.ADVANCE( LLEOTOKS, LLCURTOK );
   if  LLEOTOKS  then
      LLCURTOK.PRINTVALUE := (LLSTRINGS'RANGE => ' ');
      LLCURTOK.PRINTVALUE(1..5) := "[eof]";
      LLCURTOK.TABLEINDEX := LLLOCEOS;
   end if;
end  LLNEXTTOKEN;


procedure  LLTAKEACTION( CASEINDEX: in INTEGER )  is separate;
   -- perform the translation action proscribed in the grammar

procedure  LLMAIN  is

   LOCOFNULL: constant := 0;   -- location of null string in symbol table

   type  INTSET  is array ( 1 .. LLTABLESIZE ) of BOOLEAN;

   type  SYNCHTYPE  is
      record
         TOKEN: INTEGER;     -- index to table entry for token
         SENT: INTEGER;      -- How far in llsentential form to goto?
      end record;
```

```
type  PROD  is
   record
      LHS: INTEGER;        -- tableindex of lhs
      RHS: INTEGER;        -- index into rhsarray where rhs begins
      CARDRHS: INTEGER;    -- cardinality of rhs
      SELSET: INTSET;      -- production selection set
      CARDSEL: INTEGER;    -- cardinality of selection set
   end record;


THISRHS: INTEGER;     -- index into rhsarray
RHSARRAY:  array ( 1 .. LLRHSSIZE ) of LLRIGHT;
   -- rhs elements of productions
SYNCHDATA: array ( 0 .. LLSYNCHSIZE ) of SYNCHTYPE;
AXIOM: INTEGER;
   -- pointer to first production whose lhs is the axiom
PRODUCTIONS: array ( 1 .. LLPRODSIZE ) of PROD;


procedure  READGRAM  is      -- read grammar from disk

   CH: CHARACTER;
   LLGRAM: FILE_TYPE;      -- where grammar is stored


procedure  BUILDRIGHT( WHICHPROD: INTEGER )  is
   -- establish contents of rhs
   CHILDCOUNT: INTEGER;    -- which child in rhs is this?
   TABLEINDEX: INTEGER;
begin
   PRODUCTIONS(WHICHPROD).RHS := THISRHS + 1;
   CHILDCOUNT := 0;
   for  I  in  THISRHS+1 .. THISRHS+PRODUCTIONS(WHICHPROD).CARDRHS  loop
      if  I <= LLRHSSIZE  then
         THISRHS := THISRHS+1;
         GET( LLGRAM, CH );
         case  CH  is
            when  '1'  =>
```

```
            CHILDCOUNT := CHILDCOUNT+1;
            RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
            RHSARRAY(I).KIND := LITERAL;
            GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
         when  'a'  =>
            RHSARRAY(I).KIND := ACTION;
         when  'n'  =>
            CHILDCOUNT := CHILDCOUNT+1;
            RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
            RHSARRAY(I).KIND := NONTERMINAL;
            GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
         when  'g'  =>
            CHILDCOUNT := CHILDCOUNT+1;
            RHSARRAY(I).WHICHCHILD := CHILDCOUNT;
            RHSARRAY(I).KIND := GROUP;
            GET( LLGRAM, RHSARRAY(I).TABLEINDEX );
         when  'p'  =>
            RHSARRAY(I).KIND := PATCH;
         when others  =>
            -- the llgram table is  screwed up
            PUT( STANDARD_ERROR,
                 "*** Zuse -- Error in table file (360) ***" );
            raise  PARSING_ERROR;
      end case;
      if  END_OF_LINE( LLGRAM )  then
         RHSARRAY(I).CASEINDEX := 0;
      else
         GET( LLGRAM, RHSARRAY(I).CASEINDEX );
      end if;
      if  END_OF_LINE( LLGRAM )  then
         RHSARRAY(I).SYNCHINDEX := 0;
      else
         GET( LLGRAM, RHSARRAY(I).SYNCHINDEX );
      end if;
      SKIP_LINE( LLGRAM );
   else
      -- llgram table is screwed up
      PUT_LINE( STANDARD_ERROR,
```

```
                "*** Zuse -- Error in table file (372) ***" );
         -- This is a catastrophic error -- the grammar used to'
         -- generate the compiler probably contained errors.
         raise  PARSING_ERROR;
      end if;
   end loop;
end  BUILDRIGHT;


procedure  BUILDSELECT( WHICHPROD: INTEGER )  is
   -- build selection set
   TABLEINDEX: INTEGER;    -- Where in table can element be found?
begin
   PRODUCTIONS(WHICHPROD).SELSET := (others => FALSE);  -- empty set
   for  I  in  1 .. PRODUCTIONS(WHICHPROD).CARDSEL  loop
      GET( LLGRAM, TABLEINDEX );
      PRODUCTIONS(WHICHPROD).SELSET(TABLEINDEX) := TRUE;
   end loop;
   SKIP_LINE( LLGRAM );
end  BUILDSELECT;


begin   -- READGRAM
   OPEN( LLGRAM, IN_FILE, "TABLE" );
   -- read in symbol tables
   for  I  in  1 .. LLTABLESIZE  loop
      for  J  in  1 .. LLSTRINGLENGTH  loop
         GET( LLGRAM, LLSYMBOLTABLE(I).KEY(J) );
      end loop;
      GET( LLGRAM, CH );
      SKIP_LINE( LLGRAM );
      if  CH = 'g'  then
         LLSYMBOLTABLE(I).KIND := GROUP;
      else   -- assume ch = 1
         LLSYMBOLTABLE(I).KIND := LITERAL;
      end if;
   end loop;
   -- read in grammar
```

```
   THISRHS := 0;
   GET( LLGRAM, AXIOM );
   SKIP_LINE( LLGRAM );
   for  I  in  1 .. LLPRODSIZE  loop
      GET( LLGRAM, PRODUCTIONS(I).LHS );
      GET( LLGRAM, PRODUCTIONS(I).CARDRHS );
      GET( LLGRAM, PRODUCTIONS(I).CARDSEL );
      SKIP_LINE( LLGRAM );
      BUILDRIGHT(I);
      BUILDSELECT(I);
   end loop;
   -- now read in synchronization info
   for  I  in  1 .. LLSYNCHSIZE  loop
      GET( LLGRAM, SYNCHDATA(I).TOKEN );   -- llsymboltable location
      GET( LLGRAM, SYNCHDATA(I).SENT );   -- symbol to skip to
      SKIP_LINE( LLGRAM );
   end loop;
   CLOSE( LLGRAM );
end  READGRAM;


procedure  PARSE  is     -- parse the candidate

   LLTOP: INTEGER;       -- top of stack pointer
   LOCOFANY: INTEGER;    -- location of "any" in llsymboltable


procedure  ERASE  is
   -- Has rhs of prod been matched? If so then erase rhs.
begin
   -- only erase if at farthest point to the right in a production
   while  LLSTACK(LLSENTPTR).LASTCHILD  loop
      -- erase rhs
      LLSENTPTR := LLSTACK(LLSENTPTR).PARENT;
      if  LLSENTPTR = 0  then    -- stack is empty
         LLTOP := 0;
         LLADVANCE := FALSE;   -- don't try to advance beyond axiom
         return;
```

```
      end if;
      LLTOP := LLSTACK(LLSENTPTR).TOP;    -- lastchild of current rhs
   end loop;
end  ERASE;


procedure  TESTSYNCH;   -- forward


procedure  EXPAND  is

   -- expand nonterminal in sentential form

   WHERE: INTEGER;     -- production being examined
   OLDTOP: INTEGER;    -- top of stack ptr before expansion


function  MATCH( SENTINDEX: INTEGER )  return  INTEGER  is
   -- Does a production whose lhs is sentindex and whose selection
   -- set includes token exist?  If so, return index to that
   -- production as value of match; otherwise, set match to 0.
begin
   for  I  in  SENTINDEX .. LLPRODSIZE  loop
      if  PRODUCTIONS(I).LHS = SENTINDEX  then
         -- production has proper lhs
         if  PRODUCTIONS(I).SELSET(LLCURTOK.TABLEINDEX) or
             PRODUCTIONS(I).SELSET(LOCOFANY)  then
            return( I );    -- match found
         end if;
      else
         return( 0 );    -- no match
      end if;
   end loop;
   return( 0 );    -- no match
end  MATCH;


begin   -- EXPAND
```

```
WHERE := MATCH( LLSTACK(LLSENTPTR).DATA.TABLEINDEX );
if  WHERE /= 0  then
   -- rhs of new production will be placed in list
   if  PRODUCTIONS(WHERE).CARDRHS > 0  then
      LLADVANCE := FALSE;
      if  ( LLSTACK(LLSENTPTR).LASTCHILD and
            (LLSTACK(LLSENTPTR).PARENT > 0) )  and then
          LLSTACK(LLSTACK(LLSENTPTR).PARENT).LASTCHILD  then
         LLTOP := LLSTACK(LLSENTPTR).PARENT;
         LLSENTPTR := LLSTACK(LLSENTPTR).PARENT;
      end if;
      OLDTOP := LLTOP;
      if  LLTOP + PRODUCTIONS(WHERE).CARDRHS > LLMAXSTACK  then
         PUT_LINE( STANDARD_ERROR,
                   "*** Zuse -- Stack overflow (493) ***" );
         -- This may be fixed by increasing llmaxstack.
         LLFATAL;
      end if;
      for  I  in  1 .. PRODUCTIONS(WHERE).CARDRHS  loop
         LLTOP := LLTOP + 1;
         LLSTACK(LLTOP).PARENT := LLSENTPTR;
         -- put data into children from the selected production
         LLSTACK(LLTOP).DATA := RHSARRAY(PRODUCTIONS(WHERE).RHS+I-1);
         LLSTACK(LLTOP).LASTCHILD := FALSE;
         if  LLSTACK(LLTOP).DATA.KIND = NONTERMINAL  then
            LLSTACK(LLTOP).TOP := OLDTOP + PRODUCTIONS(WHERE).CARDRHS;
         end if;
      end loop;
      -- mark rightmost child as the last
      LLSTACK(LLTOP).LASTCHILD := TRUE;
      -- move llsentptr to the first new child
      LLSENTPTR := OLDTOP + 1;
   end if;
else
   TESTSYNCH;
end if;
end  EXPAND;
```

```
procedure TESTSYNCH is


procedure SYNCHRONIZE is
   -- synchronize token and llsentential form to recover from
   -- syntactic error
   OLDCURTOKINDEX: INTEGER;
   I: INTEGER;
begin
   PUT( STANDARD_ERROR, "*** Unexpected " );
   LLPRTTOKEN;
   PUT( STANDARD_ERROR, " found in line " );
   PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
   OLDCURTOKINDEX := LLCURTOK.TABLEINDEX;
   loop
      I := LLSTACK(LLSENTPTR).DATA.SYNCHINDEX;
      while SYNCHDATA(I).SENT /= 0 loop
         if ( (LLCURTOK.TABLEINDEX = SYNCHDATA(I).TOKEN) or
              (SYNCHDATA(I).TOKEN = LOCOFANY) ) and
            ( (SYNCHDATA(I).SENT = -1) or
              (LLSTACK(LLSENTPTR).DATA.WHICHCHILD <= SYNCHDATA(I).SENT) ) then
            if LLCURTOK.TABLEINDEX /= OLDCURTOKINDEX then
               PUT( STANDARD_ERROR, " -- skipping to " );
               LLPRTSTRING( LLCURTOK.PRINTVALUE );
               PUT( STANDARD_ERROR, " in line " );
               PUT( STANDARD_ERROR, LLCURTOK.LINENUMBER, 1 );
               PUT_LINE( STANDARD_ERROR, "." );
            end if;
            if LLSTACK(LLSENTPTR).DATA.CASEINDEX /= 0 then
               -- execute code after ";"
               LLTAKEACTION( LLSTACK(LLSENTPTR).DATA.CASEINDEX );
            end if;
            if SYNCHDATA(I).SENT = -1 then
               -- skip to rightmost node and signal reduction
               while not LLSTACK(LLSENTPTR).LASTCHILD loop
                  LLSENTPTR := LLSENTPTR + 1;
               end loop;
```

```
            else
               -- skip to correct symbol in rhs
               while  LLSTACK(LLSENTPTR).DATA.WHICHCHILD /=
                         SYNCHDATA(I).SENT  loop
                  LLSENTPTR := LLSENTPTR + 1;
               end loop;
               LLADVANCE := FALSE;
            end if;
            return;
         end if;
         I := I+1;
      end loop;
      if  LLCURTOK.TABLEINDEX = LLLOCEOS  then
         PUT_LINE( STANDARD_ERROR, " -- terminating translation." );
         raise  PARSING_ERROR;
      else
         LLNEXTTOKEN;
      end if;
   end loop;
end  SYNCHRONIZE;


begin   -- TESTSYNCH
   while  LLSTACK(LLSENTPTR).DATA.SYNCHINDEX = 0  loop
      -- no synch info there
      if  LLSTACK(LLSENTPTR).PARENT /= 0  then
         -- there really is a parent
         LLSENTPTR := LLSTACK(LLSENTPTR).PARENT;
      else
         LLFATAL;
      end if;
   end loop;
   SYNCHRONIZE;
end  TESTSYNCH;


begin   -- PARSE
   LLSENTPTR := 1;   -- initialize sentform to be axiom
```

```
LLTOP := 1;
LLSTACK(LLSENTPTR).LASTCHILD := TRUE.
LLSTACK(LLSENTPTR) PARENT := 0;
LLSTACK(LLSENTPTR).DATA.SYNCHINDEX := 0;
LLSTACK(LLSENTPTR).DATA.KIND := NONTERMINAL;
LLSTACK(LLSENTPTR).DATA.TABLEINDEX := AXIOM;
-- find location of "any" in llsymboltable
LOCOFANY := LLFIND( (1=>'a', 2=>'n', 3=>'y', others => ' '), GROUP );
-- find location of endofsource ("@") in llsymboltable
LLLOCEOS := LLFIND( (1=>'@', others => ' '), GROUP );
LLNEXTTOKEN;
while  LLTOP /= 0  loop    -- derivation isn't finished
   LLADVANCE := TRUE;    -- presume llsentptr advanced after iteration
   case  LLSTACK(LLSENTPTR).DATA.KIND  is
      when  GROUP | LITERAL  =>
         if  LLSTACK(LLSENTPTR).DATA.TABLEINDEX =
             LLCURTOK.TABLEINDEX  then
            LLSTACK(LLSENTPTR).ATTRIBUTE := LLCURTOK.ATTRIBUTE;
            LLNEXTTOKEN;
         elsif  LLSTACK(LLSENTPTR).DATA.TABLEINDEX = LOCOFNULL  then
            null;
         elsif  not LLSTACK(LLSENTPTR).LASTCHILD  then
            if  LLSTACK(LLSENTPTR + 1).DATA.KIND = PATCH  then
               LLTAKEACTION( LLSTACK(LLSENTPTR + 1).DATA.CASEINDEX );
            else
               TESTSYNCH;
            end if;
         else
            TESTSYNCH;
         end if;
      when  NONTERMINAL  =>
         EXPAND;
      when  ACTION  =>
         LLTAKEACTION( LLSTACK(LLSENTPTR).DATA.CASEINDEX );
      when  PATCH  =>
         null;
   end case;
   if  LLADVANCE  then
```

```
        -- finished with current llstack(llsentptr).
        -- move on to next node in tree
        ERASE;
        LLSENTPTR := LLSENTPTR + 1;
      end if;
   end loop;
   if  LLCURTOK.TABLEINDEX /= LLLOCEOS  then
      -- Only matched against part of candidate, which is not a sentence.
      -- Terminate parsing action.
      LLFATAL;
   end if;
end  PARSE;


begin   -- LLMAIN
   READGRAM;    -- Get the grammar from the user.
   PARSE;       -- Parse the current input stream.
end  LLMAIN;


begin  -- LL_COMPILE
   CREATE( STANDARD_ERROR, OUT_FILE, "SYS$ERROR" );   -- just in case
   LLMAIN;
   CLOSE( STANDARD_ERROR );
exception
   when  PARSING_ERROR  =>
      CLOSE( STANDARD_ERROR );
end  LL_COMPILE;
```

```
separate ( LL_COMPILE )

package body LL_TOKENS is


   -- This package is the bootstrap token stream generator for the
   -- lexical analyzer generator, ADALEX.


   subtype UPPER_CASE_LETTER is CHARACTER range 'A'..'Z';
   subtype LOWER_CASE_LETTER is CHARACTER range 'a'..'z';
   subtype DIGIT is CHARACTER range '0'..'9';


   START_OF_LINE: BOOLEAN := TRUE;
   CURRENT_CHAR: CHARACTER := ' ';
   CURRENT_LINE: INTEGER := 0;

   BUFFER_SIZE: constant := 100;

   CHAR_BUFFER: array (1 .. BUFFER_SIZE) of CHARACTER;
   LOOK_CHAR:   CHARACTER;
   CUR_BUF_NDX: INTEGER;
   TOP_BUF_NDX: INTEGER;


procedure ADVANCE( EOS:  out BOOLEAN;
                   NEXT: out LLTOKEN;
                   MORE: in  BOOLEAN := TRUE ) is

   PRINTVALUE: LLSTRINGS;
   TABLEINDEX: INTEGER;

   procedure GET_CHAR( CHAR: out CHARACTER ) is
   begin
      if END_OF_FILE(STANDARD_INPUT) then
         CHAR := ASCII.EOT;
      elsif END_OF_LINE(STANDARD_INPUT) then
```

```
      SKIP_LINE(STANDARD_INPUT);
      CHAR := ASCII.ETX;
      START_OF_LINE := TRUE;
   else
      GET(STANDARD_INPUT, CHAR);
   end if;
end  GET_CHAR;

procedure  CHAR_ADVANCE  is
begin
   if  START_OF_LINE  then
      START_OF_LINE := FALSE;
      CURRENT_LINE := CURRENT_LINE + 1;
      CUR_BUF_NDX := 0;
      TOP_BUF_NDX := 0;
      GET_CHAR(CURRENT_CHAR);
   elsif  CUR_BUF_NDX < TOP_BUF_NDX  then
      -- take char from buffer
      CUR_BUF_NDX := CUR_BUF_NDX + 1;
      CURRENT_CHAR := CHAR_BUFFER(CUR_BUF_NDX);
   else
      GET_CHAR(CURRENT_CHAR);  -- from input file
   end if;
end  CHAR_ADVANCE;

procedure  LOOK_AHEAD  is
begin
   GET_CHAR(LOOK_CHAR);
   TOP_BUF_NDX := TOP_BUF_NDX + 1;
   CHAR_BUFFER(TOP_BUF_NDX) := LOOK_CHAR;
end;

procedure  NEXT_CHARACTER  is
begin
   PRINTVALUE(1) := CURRENT_CHAR;
   CHAR_ADVANCE;
   LOOK_AHEAD;
   if  LOOK_CHAR = '''  then
```

```
      PRINTVALUE(2) := CURRENT_CHAR;
      CHAR_ADVANCE;
      PRINTVALUE(3) := CURRENT_CHAR;
      CHAR_ADVANCE;
      TABLEINDEX := LLFIND("CharLit              ", GROUP);
      NEXT.ATTRIBUTE  :=  new TREE_NODE'(CHAR,ANONYMOUS,
         (others => FALSE),FALSE,FALSE,PRINTVALUE(2));
   else
      TABLEINDEX := LLFIND(PRINTVALUE, LITERAL);
      if  TABLEINDEX = 0  then
         TABLEINDEX := LLFIND("Other                ", GROUP);
      end if;
      NEXT.ATTRIBUTE  :=  new TREE_NODE'(LIT,ANONYMOUS,
         (others => FALSE),FALSE,FALSE,PRINTVALUE);
   end if;
end  NEXT_CHARACTER;

procedure  NEXT_IDENTIFIER  is
   I: INTEGER := 1;
begin
   while  (CURRENT_CHAR in UPPER_CASE_LETTER) or
          (CURRENT_CHAR in LOWER_CASE_LETTER) or
          (CURRENT_CHAR in DIGIT) or
          (CURRENT_CHAR = '_')  loop
      if  I <= LLSTRINGLENGTH  then
         PRINTVALUE(I) := CURRENT_CHAR;
         I := I + 1;
      end if;
      CHAR_ADVANCE;
   end loop;
   TABLEINDEX := LLFIND(PRINTVALUE, LITERAL);
   if  TABLEINDEX = 0  then
      TABLEINDEX := LLFIND("Identifier          ", GROUP);
   end if;
   NEXT.ATTRIBUTE  :=  new  TREE_NODE'(IDENT,ANONYMOUS,
      (others => FALSE),FALSE,FALSE,PRINTVALUE);
end  NEXT_IDENTIFIER;
```

```
procedure  NEXT_SPEC_SYM  is
begin
   PRINTVALUE(1) := CURRENT_CHAR;
   if  CURRENT_CHAR = '.'  then
      CHAR_ADVANCE;
      if  CURRENT_CHAR = '.'  then
         PRINTVALUE(2) := CURRENT_CHAR;
         CHAR_ADVANCE;
      end if;
   elsif  CURRENT_CHAR = ':'  then
      CHAR_ADVANCE;
      if  CURRENT_CHAR = '='  then
         PRINTVALUE(2) := CURRENT_CHAR;
         CHAR_ADVANCE;
      elsif  CURRENT_CHAR = ':'  then
         LOOK_AHEAD;
         if  LOOK_CHAR = '='  then
            PRINTVALUE(2) := CURRENT_CHAR;
            CHAR_ADVANCE;
            PRINTVALUE(3) := CURRENT_CHAR;
            CHAR_ADVANCE;
         end if;
      end if;
   elsif  CURRENT_CHAR = '='  then
      CHAR_ADVANCE;
      if  CURRENT_CHAR = '>'  then
         PRINTVALUE(2) := CURRENT_CHAR;
         CHAR_ADVANCE;
      end if;
   else
      CHAR_ADVANCE;
   end if;
   TABLEINDEX := LLFIND(PRINTVALUE, LITERAL);
   if  TABLEINDEX = 0  then
      TABLEINDEX := LLFIND("Other               ", GROUP);
   end if;
   NEXT.ATTRIBUTE  :=  new TREE_NODE'(LIT,ANONYMOUS,
      (others => FALSE),FALSE,FALSE,PRINTVALUE);
```

```
end  NEXT_SPEC_SYM;

procedure  NEXT_STRING  is
   I: INTEGER := 2;
begin
   PRINTVALUE(1) := '"';
   CHAR_ADVANCE;
   while  CURRENT_CHAR /= '"'  loop
      if  I < LLSTRINGLENGTH  then
         PRINTVALUE(I) := CURRENT_CHAR;
       I := I + 1;
      end if;
   exit when  END_OF_LINE(STANDARD_INPUT);
      CHAR_ADVANCE;
   end loop;
   PRINTVALUE(I) := '"';
   CHAR_ADVANCE;
   TABLEINDEX := LLFIND("StringLit            ", GROUP);
   NEXT.ATTRIBUTE  :=  new  TREE_NODE'(STR,ANONYMOUS,
      (others => FALSE),FALSE,FALSE,PRINTVALUE);
end  NEXT_STRING;

begin   -- ADVANCE
   PRINTVALUE := "                         ";
   -- Skip white space and comments
   while  (CURRENT_CHAR = ASCII.ETX) or
          (CURRENT_CHAR = ASCII.HT) or
          (CURRENT_CHAR = ' ') or
          (CURRENT_CHAR = '-')  loop
      if  CURRENT_CHAR = '-'  then
         LOOK_AHEAD;
         exit when  LOOK_CHAR /= '-';
         SKIP_LINE(STANDARD_INPUT);
         START_OF_LINE := TRUE;
      end if;
      CHAR_ADVANCE;
   end loop;
   if  CURRENT_CHAR = ASCII.EOT  then
```

```
         EOS := TRUE;
      elsif  CURRENT_CHAR = '"'  then
         NEXT_STRING;
      elsif  CURRENT_CHAR = '''  then
         NEXT_CHARACTER;
      elsif  (CURRENT_CHAR in UPPER_CASE_LETTER) or
             (CURRENT_CHAR in LOWER_CASE_LETTER)  then
         NEXT_IDENTIFIER;
      else
         NEXT_SPEC_SYM;
      end if;
      NEXT.PRINTVALUE := PRINTVALUE;
      NEXT.TABLEINDEX := TABLEINDEX;
      NEXT.LINENUMBER := CURRENT_LINE;
   end  ADVANCE;


end  LL_TOKENS;
```

```
--
--
--                ADALEX  SUPPORT  PACKAGE  SPECIFICATION
--
--
--  This package contains all the supporting translation routines for
--  the Adalex translator.  These procedures and functions are called
--  from the parsing actions specified in the translation grammar.
--
--
--  Associated packages, procedures, and files:
--
--    o  The body of this package is defined in file LL_SUP_BODY.ADA
--
--    o  The Adalex translation grammar is defined in file ADALEX.GRM
--
--    o  The parsing actions are included in procedure LLTAKEACTION in
--       file LL_ACTIONS.ADA
--
--    o  Declarations for data structures defined in the grammar are
--       included in package LL_DECLARATIONS in file LL_DECLS.ADA
--
--

with  LL_DECLARATIONS;

package  LL_SUPPORT  is

   use  LL_DECLARATIONS;

   PATTERN_TABLE_FULL: exception;
      --  Raised if the pattern table overflows.

   function  ALTERNATE ( LEFT, RIGHT: in LLATTRIBUTE ) return  LLATTRIBUTE;
      --  Form the alternation of two patterns.

   function  CHAR_RANGE ( START_CH, END_CH: in LLATTRIBUTE )
      return  LLATTRIBUTE;
```

```
   --  Form a character or range pattern.

procedure  COMPLETE_PATTERNS;
   --  Complete the construction of all the patterns defined.

function  CONCATENATE ( LEFT, RIGHT: in LLATTRIBUTE ) return  LLATTRIBUTE;
   --  Concatenate two patterns.

function  CVT_ASCII ( NAME: in LLATTRIBUTE ) return  LLATTRIBUTE;
   --  Convert an ASCII character name into a character pattern.

function  CVT_STRING ( LIT: in LLATTRIBUTE ) return  LLATTRIBUTE;
   --  Convert a literal string into a pattern.

procedure  EMIT_ADVANCE_HDR;
   --  Emit the beginning part of the procedure ADVANCE.

procedure  EMIT_ADVANCE_TLR;
   --  Emit the end of the procedure ADVANCE.

procedure  EMIT_PKG_DECLS;
   --  Emit the declarations for the generated package body.

procedure  EMIT_SCAN_PROC;
   --  Generate the pattern-matching code for all referenced patterns.

procedure  EMIT_TOKEN ( TOKEN: in LLATTRIBUTE );
   --  Emit an identifier or literal token value.

procedure  INCLUDE_PATTERN( PAT_ID: in LLATTRIBUTE );
   --  Include the referenced pattern for code generation.

function  LOOK_AHEAD ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE;
   --  Create a look-ahead pattern.

function  OPTION ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE;
   --  Form an optional pattern.
```

```
   function  REPEAT ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE;
      --  Form a repetition pattern.

   procedure  STORE_PATTERN ( PAT_ID, PAT: in LLATTRIBUTE );
      --  Store a pattern definition in the pattern table.

end  LL_SUPPORT;
```

```
--
--
--                    ADALEX  SUPPORT  PACKAGE  BODY
--
--
--  This package contains all the supporting translation routines for
--  the Adalex translator.  These procedures and functions are called
--  from the parsing actions specified in the translation grammar.
--
--
--  Associated packages, procedures, and files:
--
--     o  The specification for this package is defined in file
--        LL_SUP_SPEC.ADA
--
--     o  The Adalex translation grammar is defined in file ADALEX.GRM
--
--     o  The parsing actions are included in procedure LLTAKEACTION in
--        file LL_ACTIONS.ADA
--
--     o  Declarations for data structures defined in the grammar are
--        included in package LL_DECLARATIONS in file LL_DECLS.ADA
--
--

with  LL_DECLARATIONS, TEXT_IO, INTEGER_TEXT_IO;

package body  LL_SUPPORT  is

   use  LL_DECLARATIONS, TEXT_IO, INTEGER_TEXT_IO;

   EMPTY_PATTERN: constant LLATTRIBUTE :=
      new TREE_NODE'(EMPTY,ANONYMOUS,(others => FALSE),TRUE,FALSE);

   OUTPUT_LINE_LIMIT: constant := 60;
      --  If an output line exceeds this limit, start a new line.

   PATTERN_TABLE_SIZE: constant := 100;
```

```
   --  The maximum number of entries in the pattern table.

CUR_TABLE_ENTRIES: INTEGER range 0 .. PATTERN_TABLE_SIZE := 0;
   --  Current number of pattern table entries.
   --  Updated by procedure STORE_PATTERN.

EMITTED_CHARS: INTEGER := 0;
   --  The number of characters emitted on the current output line.

LEXICON: LLATTRIBUTE := null;
   --  Pattern constructed for code generation.
   --  Created by calls to procedure INCLUDE_PATTERN.
   --  Consumed by procedure EMIT_SCAN_PROC.

PATTERN_TABLE: array (1 .. PATTERN_TABLE_SIZE) of LLATTRIBUTE;
   --  Table containing all defined patterns in alphabetical order.
   --  Updated by procedure STORE_PATTERN.

ROOT_PATTERN_NAME: LLSTRINGS;
   --  Holds the name of the current pattern being completed.
   --  Used to check for recursive references.


procedure  COMPLETE_PAT ( PAT: in out LLATTRIBUTE );
   --  Complete the construction of an arbitrary pattern.

procedure  EMIT_PATTERN_NAME ( FILE: in FILE_TYPE; NAME: in LLSTRINGS );
   --  Write the name of a pattern to a specified file.

procedure  EMIT_PATTERN_NAME ( NAME: in LLSTRINGS );
   --  Write the name of a pattern to the standard output file.

function  LOOK_UP_PATTERN ( PAT_ID: in LLATTRIBUTE ) return  INTEGER;
   --  Return the index of the named pattern in the pattern table.


function  ALTERNATE ( LEFT, RIGHT: in LLATTRIBUTE ) return  LLATTRIBUTE  is
```

```
    -- Form the alternation of two patterns.

    NEW_LEFT, NEW_RIGHT: LLATTRIBUTE;

    function  MERGE_RANGES ( LEFT, RIGHT: in LLATTRIBUTE )
       return  LLATTRIBUTE  is
       --  Merge the selection sets of two range nodes into a single node.
       SEL_SET: SELECTION_SET;
    begin
       for  CH in SELECTION SET'RANGE  loop
          SEL_SET(CH) := LEFT.SEL_SET(CH) or RIGHT.SEL_SET(CH);
       end loop;
       return  new TREE_NODE'(RNG,LEFT.NAME,SEL_SET,FALSE,FALSE);
    end  MERGE_RANGES;

 begin  -- ALTERNATE(LEFT,RIGHT)
    --  Form the alternation of two patterns.
    --  Create an alternation node if the right term is not empty.
    if  RIGHT = null or else RIGHT.VARIANT = BAD then
       return  LEFT;
    elsif  LEFT.VARIANT = BAD  then
       return  RIGHT;
    end if;
    if  LEFT.VARIANT = ALT  then
       if RIGHT.VARIANT = ALT  then
          if  LEFT.NAME = ANONYMOUS then
             NEW_LEFT  := LEFT.LEFT;
             NEW_RIGHT  := ALTERNATE(LEFT.RIGHT,RIGHT);
          elsif  RIGHT.NAME = ANONYMOUS  then
             NEW_LEFT  := RIGHT.LEFT;
             NEW_RIGHT  := ALTERNATE(RIGHT.RIGHT,LEFT);
          else
             NEW_LEFT  := new TREE_NODE'(LEFT.LEFT.all);
             NEW_LEFT.NAME  := LEFT.NAME;
             NEW_RIGHT  := new TREE_NODE'(LEFT.RIGHT.all);
             NEW_RIGHT.NAME  := LEFT.NAME;
             NEW_RIGHT  := ALTERNATE(NEW_RIGHT,RIGHT);
          end if;
```

```
      else
         NEW_LEFT := RIGHT;
         NEW_RIGHT := LEFT;
      end if;
   else
      NEW_LEFT := LEFT;
      NEW_RIGHT := RIGHT;
   end if;
   if  NEW_LEFT.VARIANT = RNG  then
      if  NEW_RIGHT.VARIANT = RNG
          and NEW_LEFT.NAME = NEW_RIGHT.NAME  then
         return  MERGE_RANGES(NEW_LEFT,NEW_RIGHT);
      elsif  (NEW_RIGHT.VARIANT = ALT and NEW_RIGHT.NAME = ANONYMOUS)
             and then NEW_RIGHT.LEFT.VARIANT = RNG
             and then NEW_LEFT.NAME = NEW_RIGHT.LEFT.NAME  then
         return  new TREE_NODE'(ALT,ANONYMOUS,(others => FALSE),
                        FALSE,FALSE,MERGE_RANGES(NEW_LEFT,NEW_RIGHT.LEFT),
                        NEW_RIGHT.RIGHT );
      else
         return  new TREE_NODE'(ALT,ANONYMOUS,(others => FALSE),
                                FALSE,FALSE,NEW_LEFT,NEW_RIGHT);
      end if;
   elsif  NEW_RIGHT.VARIANT = RNG  then
      --  keep ranges on the left for convenience
      return  new TREE_NODE'(ALT,ANONYMOUS,(others => FALSE),
                             FALSE,FALSE,NEW_RIGHT,NEW_LEFT);
   else
      return  new TREE_NODE'(ALT,ANONYMOUS,(others => FALSE),
                             FALSE,FALSE,NEW_LEFT,NEW_RIGHT);
   end if;
end  ALTERNATE;


function  CHAR_RANGE ( START_CH, END_CH: in LLATTRIBUTE )
   return  LLATTRIBUTE  is
   --  Form a character or range pattern.
   --  Create a range node for a single character or range expression.
   RESULT: LLATTRIBUTE;
```

```
begin
   RESULT := new TREE_NODE'(RNG,ANONYMOUS,(others => FALSE),FALSE,FALSE);
   if  END_CH = null  then
      --  the pattern is a single character
      RESULT.SEL_SET(START_CH.CHAR_VAL) := TRUE;
   else
      --  the pattern is a range expression
      for CH in START_CH.CHAR_VAL .. END_CH.CHAR_VAL loop
         RESULT.SEL_SET(CH) := TRUE;
      end loop;
   end if;
   return  RESULT;
end  CHAR_RANGE;


procedure  COMPLETE_PAT ( PAT: in out LLATTRIBUTE )  is

   --  Complete the construction of an arbitrary pattern.

   N: INTEGER range 0 .. PATTERN_TABLE_SIZE;

   procedure  COMPLETE_CONCAT ( PAT: in out LLATTRIBUTE );
      --  Complete the construction of a concatenation node.

   procedure  COMPLETE_OPT ( PAT: in out LLATTRIBUTE );
      --  Complete the construction of an optional pattern.

   procedure  COMPLETE_ALT ( PAT: in out LLATTRIBUTE )  is

      --  Complete the construction of an alternation pattern.
      --  Maintain the pattern in normal form for code generation.
      --  Convert patterns with empty alternatives into option patterns.
      --  Convert ambiguous patterns into equivalent unambiguous patterns.

      NAME: LLSTRINGS;
      INTERSECT: BOOLEAN := FALSE;

      function  RESTRICT ( PAT: in LLATTRIBUTE; SUBSET: in SELECTION_SET )
```

```
   return LLATTRIBUTE  is
   --  Return the subset of a pattern that is restricted to a
   --  specified selection set.
   EMPTY: BOOLEAN := TRUE;
   NEW_PAT, NEW_LEFT, NEW_RIGHT: LLATTRIBUTE;
   NEW_SET: SELECTION_SET;
begin
   case PAT.VARIANT is
      when  ALT  =>
         --  Restrict both alternatives
         NEW_LEFT := RESTRICT(PAT.LEFT,SUBSET);
         NEW_RIGHT := RESTRICT(PAT.RIGHT,SUBSET);
         if NEW_LEFT = EMPTY_PATTERN then
            if PAT.NAME /= ANONYMOUS then
               NEW_RIGHT.NAME := PAT.NAME;
            end if;
            return NEW_RIGHT;
         elsif NEW_RIGHT = EMPTY_PATTERN then
            if PAT.NAME /= ANONYMOUS then
               NEW_LEFT.NAME := PAT.NAME;
            end if;
            return NEW_LEFT;
         else
            NEW_PAT := ALTERNATE(NEW_LEFT,NEW_RIGHT);
            NEW_PAT.NAME := PAT.NAME;
            COMPLETE_PAT(NEW_PAT);
            return NEW_PAT;
         end if;
      when  CAT  =>
         --  Restrict the left component.
         NEW_LEFT := RESTRICT(PAT.LEFT,SUBSET);
         if NEW_LEFT = EMPTY_PATTERN then
            return EMPTY_PATTERN;
         else
            NEW_PAT := CONCATENATE(NEW_LEFT,PAT.RIGHT);
            NEW_PAT.NAME := PAT.NAME;
            COMPLETE_CONCAT(NEW_PAT);
            return NEW_PAT;
```

```
            end if;
         when  OPT | REP  =>
            --  Restrict the optional or repeated pattern
            NEW_PAT := RESTRICT(PAT.EXPR,SUBSET);
            if NEW_PAT = EMPTY_PATTERN then
               return EMPTY_PATTERN;
            elsif PAT.VARIANT = OPT then
               NEW_PAT := OPTION(NEW_PAT);
               NEW_PAT.NAME := PAT.NAME;
               COMPLETE_OPT(NEW_PAT);
               return NEW_PAT;
            else  -- PAT.VARIANT = REP
               NEW_PAT := OPTION( CONCATENATE(NEW_PAT,PAT) );
               NEW_PAT.NAME := PAT.NAME;
               COMPLETE_OPT(NEW_PAT);
               return NEW_PAT;
            end if;
         when  RNG  =>
            --  Restrict a simple range
            for CH in SELECTION_SET'RANGE loop
               NEW_SET(CH) := SUBSET(CH) and PAT.SEL_SET(CH);
               EMPTY := EMPTY and not NEW_SET(CH);
            end loop;
            if EMPTY then
               return EMPTY_PATTERN;
            else
               return  new TREE_NODE'(RNG,PAT.NAME,NEW_SET,FALSE,FALSE);
            end if;
         when others  =>
            --  No other kinds of patterns should show up here.
            return BAD_PATTERN;
      end case;
   end  RESTRICT;

   function  TAIL ( PAT: in LLATTRIBUTE; SUBSET: in SELECTION_SET )
      return LLATTRIBUTE  is
      -- Return the tail of pattern PAT with the selection set SUBSET.
      LEFT_SET, RIGHT_SET: SELECTION_SET;
```

```
   NEW_PAT, NEW_LEFT, NEW_RIGHT: LLATTRIBUTE;
begin
   case PAT.VARIANT is
      when  ALT  =>
         --  Combine the tails of the two alternatives
         for CH in SELECTION_SET'RANGE loop
            LEFT_SET(CH) := PAT.LEFT.SEL_SET(CH) and SUBSET(CH);
            RIGHT_SET(CH) := PAT.RIGHT.SEL_SET(CH) and SUBSET(CH);
         end loop;
         NEW_PAT := ALTERNATE( TAIL(PAT.LEFT,LEFT_SET),
                               TAIL(PAT.RIGHT,RIGHT_SET) );
      when  CAT  =>
         case PAT.LEFT.VARIANT is
            when  ALT  =>
               --  Convert pattern (A|B)C into AC|BC then find tail.
               NEW_LEFT := CONCATENATE(PAT.LEFT.LEFT,PAT.RIGHT);
               COMPLETE_CONCAT(NEW_LEFT);
               NEW_RIGHT := CONCATENATE(PAT.LEFT.RIGHT,PAT.RIGHT);
               COMPLETE_CONCAT(NEW_RIGHT);
               for CH in SELECTION_SET'RANGE loop
                  LEFT_SET(CH) := SUBSET(CH) and
                                  NEW_LEFT.SEL_SET(CH);
                  RIGHT_SET(CH) := SUBSET(CH) and
                                   NEW_RIGHT.SEL_SET(CH);
               end loop;
               NEW_PAT := ALTERNATE( TAIL(NEW_LEFT,LEFT_SET),
                                     TAIL(NEW_RIGHT,RIGHT_SET) );
            when  OPT | REP  =>
               --  Convert pattern [A]B into AB|B then find tail.
               for CH in SELECTION_SET'RANGE loop
                  LEFT_SET(CH) := SUBSET(CH) and
                                  PAT.LEFT.SEL_SET(CH);
                  RIGHT_SET(CH) := SUBSET(CH) and
                                   not PAT.LEFT.SEL_SET(CH);
               end loop;
               if PAT.LEFT.VARIANT = OPT then
                  NEW_LEFT := CONCATENATE(
                                 TAIL(PAT.LEFT.EXPR,LEFT_SET),
```

```
                                  PAT.RIGHT );
                  else   -- PAT.LEFT.VARIANT = REP
                     NEW_LEFT := CONCATENATE(
                                  TAIL(PAT.LEFT.EXPR,LEFT_SET),
                                  PAT );
                  end if;
                  COMPLETE_CONCAT(NEW_LEFT);
                  NEW_PAT := ALTERNATE( NEW_LEFT,
                                      TAIL(PAT.RIGHT,RIGHT_SET) );
               when  RNG  =>
                  --  This one is easy.
                  return new TREE_NODE'(PAT.RIGHT.all);
               when others  =>
                  --  No other kinds of patterns should show up here.
                  return BAD_PATTERN;
            end case;
         when  RNG  =>
            --  This one is easy too.
            return EMPTY_PATTERN;
         when others  =>
            --  No other kinds of patterns should show up here.
            return BAD_PATTERN;
      end case;
      COMPLETE_PAT(NEW_PAT);
      return NEW_PAT;
   end  TAIL;

   procedure  RESOLVE_AMBIGUITY ( PAT: in out LLATTRIBUTE )  is
      --  Resolve the ambiguity in an alternative pattern.
      --  Ambiguity arises when two alternatives have overlapping
      --  selection sets.  Several transformations are applied here
      --  to create an equivalent pattern without any overlap.
      LEFT, RIGHT, MIDDLE : LLATTRIBUTE;
      LEFT_TAIL, RIGHT_TAIL: LLATTRIBUTE;
      LEFT_SET, MIDDLE_SET, RIGHT_SET: SELECTION_SET;
      NAME: LLSTRINGS := PAT.NAME;
   begin
      --  Separate the selection sets into left, middle, and right sets.
```

```
for CH in SELECTION_SET'RANGE loop
   LEFT_SET(CH) := PAT.LEFT.SEL_SET(CH) and
                         not PAT.RIGHT.SEL_SET(CH);
   RIGHT_SET(CH) := PAT.RIGHT.SEL_SET(CH) and
                          not PAT.LEFT.SEL_SET(CH);
   MIDDLE_SET(CH) := PAT.LEFT.SEL_SET(CH) and
                           PAT.RIGHT.SEL_SET(CH);
end loop;
--  Construct a new pattern for the overlapping middle part.
LEFT := RESTRICT(PAT.LEFT,MIDDLE_SET);
RIGHT := RESTRICT(PAT.RIGHT,MIDDLE_SET);
if  LEFT.VARIANT = ALT  then
   MIDDLE := new TREE_NODE'(LEFT.RIGHT.all);
   MIDDLE.NAME := LEFT.NAME;
   RIGHT := ALTERNATE(MIDDLE,RIGHT);
   COMPLETE_PAT(RIGHT);
   MIDDLE := new TREE_NODE'(LEFT.LEFT.all);
   MIDDLE.NAME := LEFT.NAME;
   MIDDLE := ALTERNATE(MIDDLE,RIGHT);
elsif  RIGHT.VARIANT = ALT  then
   MIDDLE := new TREE_NODE'(RIGHT.LEFT.all);
   MIDDLE.NAME := RIGHT.NAME;
   LEFT := ALTERNATE(LEFT,MIDDLE);
   COMPLETE_PAT(LEFT);
   MIDDLE := new TREE_NODE'(RIGHT.RIGHT.all);
   MIDDLE.NAME := RIGHT.NAME;
   MIDDLE := ALTERNATE(LEFT,MIDDLE);
else
   LEFT_TAIL := TAIL(LEFT,MIDDLE_SET);
   RIGHT_TAIL := TAIL(RIGHT,MIDDLE_SET);
   if LEFT_TAIL = EMPTY_PATTERN then
      RIGHT_TAIL := OPTION(RIGHT_TAIL);
      RIGHT_TAIL.NAME := RIGHT.NAME;
      COMPLETE_OPT(RIGHT_TAIL);
      MIDDLE := CONCATENATE( new
                   TREE_NODE'(RNG,LEFT.NAME,MIDDLE_SET,FALSE,FALSE),
                   RIGHT_TAIL);
   elsif RIGHT_TAIL = EMPTY_PATTERN then
```

```
        LEFT_TAIL := OPTION(LEFT_TAIL);
        LEFT_TAIL.NAME := LEFT.NAME;
        COMPLETE_OPT(LEFT_TAIL);
        MIDDLE := CONCATENATE( new TREE_NODE'(RNG,RIGHT.NAME,
                     MIDDLE_SET,FALSE,FALSE), LEFT_TAIL );
     elsif LEFT.VARIANT = CAT and then
        (LEFT.LEFT.VARIANT = RNG and LEFT.RIGHT.VARIANT = OPT) then
        RIGHT_TAIL.NAME := RIGHT.NAME;
        if LEFT.NAME = ANONYMOUS then
           MIDDLE := CONCATENATE( new TREE_NODE'(RNG,LEFT.LEFT.NAME,
                        MIDDLE_SET,FALSE,FALSE),
                        ALTERNATE(LEFT.RIGHT,RIGHT_TAIL) );
        else
           MIDDLE := CONCATENATE( new TREE_NODE'(RNG,LEFT.NAME,
                        MIDDLE_SET,FALSE,FALSE),
                        ALTERNATE(LEFT_TAIL,RIGHT_TAIL) );
        end if;
     elsif RIGHT.VARIANT = CAT and then
        (RIGHT.LEFT.VARIANT = RNG and RIGHT.RIGHT.VARIANT = OPT) then
        LEFT_TAIL.NAME := LEFT.NAME;
        if RIGHT.NAME = ANONYMOUS then
           MIDDLE := CONCATENATE( new TREE_NODE'(RNG,RIGHT.LEFT.NAME,
                        MIDDLE_SET,FALSE,FALSE),
                        ALTERNATE(LEFT_TAIL,RIGHT.RIGHT) );
        else
           MIDDLE := CONCATENATE( new TREE_NODE'(RNG,RIGHT.NAME,
                        MIDDLE_SET,FALSE,FALSE),
                        ALTERNATE(LEFT_TAIL,RIGHT_TAIL) );
        end if;
     else
        LEFT_TAIL.NAME := LEFT.NAME;
        RIGHT_TAIL.NAME := RIGHT.NAME;
        MIDDLE := CONCATENATE( new TREE_NODE'(RNG,ANONYMOUS,
                     MIDDLE_SET,FALSE,FALSE),
                     ALTERNATE(LEFT_TAIL,RIGHT_TAIL) );
     end if;
  end if;
  COMPLETE_PAT(MIDDLE);
```

```
      --  Restrict the non-overlapping parts of the pattern to their
      --  respective subsets and reconstruct the complete pattern.
      LEFT := RESTRICT(PAT.LEFT,LEFT_SET);
      RIGHT := RESTRICT(PAT.RIGHT,RIGHT_SET);
      if LEFT = EMPTY_PATTERN then
         if RIGHT = EMPTY_PATTERN then
            PAT := MIDDLE;
         else
            PAT := ALTERNATE(MIDDLE,RIGHT);
            COMPLETE_PAT(PAT);
         end if;
      elsif RIGHT = EMPTY_PATTERN then
         PAT := ALTERNATE(LEFT,MIDDLE);
         COMPLETE_PAT(PAT);
      else
         PAT := ALTERNATE( LEFT, ALTERNATE(MIDDLE,RIGHT) );
         COMPLETE_PAT(PAT);
      end if;
      PAT.NAME := NAME;
   end  RESOLVE_AMBIGUITY;

begin   --  COMPLETE_ALT(PAT)
   --  Complete the construction of an alternation pattern.
   --  Make the remaining alternative optional if one side is empty.
   if  PAT.LEFT = EMPTY_PATTERN  then
      if  PAT.RIGHT = EMPTY_PATTERN  then
         PAT := EMPTY_PATTERN;
      else
         PAT := OPTION(PAT.RIGHT);
         COMPLETE_OPT(PAT);
      end if;
   elsif  PAT.RIGHT = EMPTY_PATTERN  then
      PAT := OPTION(PAT.LEFT);
      COMPLETE_OPT(PAT);
   else
      COMPLETE_PAT(PAT.LEFT);
      COMPLETE_PAT(PAT.RIGHT);
      --  Combine the two selection sets and see if they overlap.
```

```
      for  CH in SELECTION_SET'RANGE  loop
         PAT.SEL_SET(CH) := PAT.LEFT.SEL_SET(CH) or PAT.RIGHT.SEL_SET(CH);
         INTERSECT := INTERSECT or
                      (PAT.LEFT.SEL_SET(CH) and PAT.RIGHT.SEL_SET(CH));
      end loop;
      if  INTERSECT  then
         RESOLVE_AMBIGUITY(PAT);
      else
         --  If either alternative is optional simplify the pattern.
         PAT.NULLABLE := PAT.LEFT.NULLABLE or PAT.RIGHT.NULLABLE;
         if  PAT.LEFT.VARIANT = OPT  then
            NAME := PAT.LEFT.NAME;
            PAT.LEFT := new TREE_NODE'(PAT.LEFT.EXPR.all);
            PAT.LEFT.NAME := NAME;
         end if;
         if  PAT.RIGHT.VARIANT = OPT  then
            NAME := PAT.RIGHT.NAME;
            PAT.RIGHT := new TREE_NODE'(PAT.RIGHT.EXPR.all);
            PAT.RIGHT.NAME := NAME;
         end if;
      end if;
   end if;
end  COMPLETE_ALT;

procedure  COMPLETE_CONCAT ( PAT: in out LLATTRIBUTE )  is
   --  Complete the construction of a concatenation node.
   --  Maintain the pattern in normal form for code generation.
   SEL_SET: SELECTION_SET;
begin
   if  PAT.LEFT = EMPTY_PATTERN  then
      PAT := PAT.RIGHT;
      COMPLETE_PAT(PAT);
   else
      COMPLETE_PAT(PAT.LEFT);
      COMPLETE_PAT(PAT.RIGHT);
      --  Make concatenations right associative for code generation.
      while  PAT.LEFT.VARIANT = CAT  loop
         PAT.RIGHT := CONCATENATE(PAT.LEFT.RIGHT,PAT.RIGHT);
```

```
         COMPLETE_CONCAT(PAT.RIGHT);
         PAT.LEFT := PAT.LEFT.LEFT;
      end loop;
      if  PAT.LEFT.NULLABLE  then
         case  PAT.LEFT.VARIANT  is
            when  ALT  =>
               PAT.LEFT := new TREE_NODE'(PAT.LEFT.all);
               PAT.LEFT.NULLABLE := FALSE;
               PAT := ALTERNATE( CONCATENATE(PAT.LEFT,PAT.RIGHT),
                                 PAT.RIGHT );
               COMPLETE_ALT(PAT);
            when  REP  =>
               for  CH in SELECTION_SET'RANGE  loop
                  PAT.SEL_SET(CH) := PAT.LEFT.SEL_SET(CH) or
                                     PAT.RIGHT.SEL_SET(CH);
               end loop;
               PAT.NULLABLE := PAT.RIGHT.NULLABLE;
            when  OPT  =>
               PAT := ALTERNATE( CONCATENATE(PAT.LEFT.EXPR,PAT.RIGHT),
                                 PAT.RIGHT );
               COMPLETE_ALT(PAT);
            when others  =>
               --  No other kinds of patterns should show up here.
               PAT := BAD_PATTERN;
         end case;
      else
         PAT.SEL_SET := PAT.LEFT.SEL_SET;
      end if;
   end if;
end  COMPLETE_CONCAT;

procedure  COMPLETE_OPT ( PAT: in out LLATTRIBUTE )  is
   --  Complete the construction of an optional pattern.
   --  Maintain the pattern in normal form for code generation.
   --  Fill in the selection set and make the pattern nullable.
   NAME: LLSTRINGS;
begin
   COMPLETE_PAT(PAT.EXPR);
```

```
      case  PAT.EXPR.VARIANT  is
         when  ALT  =>
            NAME := PAT.NAME;
            PAT := PAT.EXPR;
            PAT.NAME := NAME;
         when  CAT | RNG  =>
            PAT.SEL_SET := PAT.EXPR.SEL_SET;
         when others  =>
            --  No other kinds of patterns should show up here.
            PAT := BAD_PATTERN;
            return;
      end case;
      PAT.NULLABLE := TRUE;
   end  COMPLETE_OPT;

begin   --  COMPLETE_PAT(PAT)
   --  Complete the construction of an arbitrary pattern.
   if  PAT.SEL_SET = EMPTY_PATTERN.SEL_SET  then
      --  It has not been completed yet.
      case  PAT.VARIANT  is
         when  ALT  =>
            COMPLETE_ALT(PAT);
         when  BAD  =>
            null;
         when  CAT  =>
            COMPLETE_CONCAT(PAT);
         when  IDENT  =>
            --  Check for a recursive pattern reference.
            if  PAT.STRING_VAL = ROOT_PATTERN_NAME  then
               PUT(STANDARD_ERROR,"*** Pattern """);
               EMIT_PATTERN_NAME(STANDARD_ERROR,PAT.STRING_VAL);
               PUT(STANDARD_ERROR,""" on line ");
               PUT(STANDARD_ERROR,0,1);
               PUT_LINE(STANDARD_ERROR," is defined recursively.");
               PAT := EMPTY_PATTERN;
            else
               --  Pick up the definition from the pattern table.
               N := LOOK_UP_PATTERN(PAT);
```

```
            if  N = 0  then
               PUT(STANDARD_ERROR,"*** Pattern """);
               EMIT_PATTERN_NAME(STANDARD_ERROR,PAT.STRING_VAL);
               PUT(STANDARD_ERROR,""" refered to in line ");
               PUT(STANDARD_ERROR,0,1);
               PUT_LINE(STANDARD_ERROR," is not defined.");
               PAT := EMPTY_PATTERN;
            else
               PAT := PATTERN_TABLE(N);
               COMPLETE_PAT(PAT);
            end if;
         end if;
      when  LOOK  =>
         COMPLETE_PAT(PAT.LEFT);
         COMPLETE_PAT(PAT.RIGHT);
         for  CH in SELECTION_SET'RANGE  loop
            PAT.SEL_SET(CH) := PAT.LEFT.SEL_SET(CH) or
                               PAT.RIGHT.SEL_SET(CH);
         end loop;
      when  OPT  =>
         COMPLETE_OPT(PAT);
      when  REP  =>
         COMPLETE_PAT(PAT.EXPR);
         PAT.SEL_SET := PAT.EXPR.SEL_SET;
      when others  =>
         --  No other kinds of patterns should show up here.
         PAT := BAD_PATTERN;
    end case;
  end if;
end  COMPLETE_PAT;


procedure  COMPLETE_PATTERNS  is
  -- Complete the construction of all the patterns defined.
begin
  for  I in 1 .. CUR_TABLE_ENTRIES  loop
    ROOT_PATTERN_NAME := PATTERN_TABLE(I).NAME;
    COMPLETE_PAT( PATTERN_TABLE(I) );
```

```
   end loop;
end  COMPLETE_PATTERNS;


function  CONCATENATE ( LEFT, RIGHT: in LLATTRIBUTE )
   return  LLATTRIBUTE  is
   --  Concatenate two patterns.
   --  Create a concatenation node if the right term is not empty.
begin
   if  RIGHT = null or else RIGHT.VARIANT = BAD  then
      return  LEFT;
   elsif  LEFT.VARIANT = BAD  then
      return  RIGHT;
   else
      return  new TREE_NODE'(CAT,ANONYMOUS,(others => FALSE),
                             FALSE,FALSE,LEFT,RIGHT);
   end if;
end  CONCATENATE;


function  CVT_ASCII ( NAME: in LLATTRIBUTE )
   return  LLATTRIBUTE  is
   --  Convert an ASCII character name into a character pattern.
   CH: CHARACTER;
begin
   if  NAME.STRING_VAL(1..4) = "BEL "  then
      CH := ASCII.BEL;
   elsif  NAME.STRING_VAL(1..3) = "BS "  then
      CH := ASCII.BS;
   elsif  NAME.STRING_VAL(1..3) = "HT "  then
      CH := ASCII.HT;
   elsif  NAME.STRING_VAL(1..3) = "LF "  then
      CH := ASCII.LF;
   elsif  NAME.STRING_VAL(1..3) = "VT "  then
      CH := ASCII.VT;
   elsif  NAME.STRING_VAL(1..3) = "FF "  then
      CH := ASCII.FF;
   elsif  NAME.STRING_VAL(1..3) = "CR "  then
```

```
      CH := ASCII.CR;
   elsif  NAME.STRING_VAL(1..4) = "ESC "  then
      CH := ASCII.ESC;
   elsif  NAME.STRING_VAL(1..4) = "DEL "  then
      CH := ASCII.DEL;
   else
      CH := ASCII.NUL;
   end if;
   return  new TREE_NODE'(CHAR,ANONYMOUS,(others => FALSE),
                           FALSE,FALSE,CH);
end  CVT_ASCII;


function  CVT_STRING ( LIT: in LLATTRIBUTE ) return  LLATTRIBUTE  is
   --  Convert a literal string into a pattern.
   --  The string "ABC" becomes the concatenation 'A' 'B' 'C'.
   LEFT, RIGHT: LLATTRIBUTE;
begin
   if LIT.STRING_VAL(2) = '"' then
      return EMPTY_PATTERN;
   else
      LEFT := new TREE_NODE'(RNG,ANONYMOUS,(others => FALSE),FALSE,FALSE);
      LEFT.SEL_SET(LIT.STRING_VAL(2)) := TRUE;
      for  I in 3 .. LLSTRINGS'LAST  loop
      exit when LIT.STRING_VAL(I) = '"';
         RIGHT := new TREE_NODE'(RNG,ANONYMOUS,(others => FALSE),
                                 FALSE,FALSE);
         RIGHT.SEL_SET(LIT.STRING_VAL(I)) := TRUE;
         LEFT := CONCATENATE(LEFT,RIGHT);
      end loop;
      return LEFT;
   end if;
end  CVT_STRING;


procedure  EMIT_ADVANCE_HDR  is
   --  Emit the beginning of the definition of procedure ADVANCE.
begin
```

```
   NEW_LINE;
   PUT_LINE("  procedure ADVANCE(EOS: out BOOLEAN;");
   PUT_LINE("    NEXT: out TOKEN;");
   PUT_LINE("    MORE: in BOOLEAN := TRUE) is");
   PUT_LINE("  begin");
   PUT_LINE("    EOS := FALSE;");
   PUT_LINE("    loop");
   PUT_LINE("      SCAN_PATTERN;");
   PUT_LINE("      case CUR_PATTERN is");
   PUT_LINE("        when END_OF_INPUT =>");
   PUT_LINE("          EOS := TRUE;");
   PUT_LINE("          return;");
   PUT_LINE("        when END_OF_LINE => null;");
end  EMIT_ADVANCE_HDR;


procedure  EMIT_ADVANCE_TLR  is
   --  Emit the end of the definition of procedure ADVANCE.
begin
   PUT_LINE("      end case;");
   PUT_LINE("    end loop;");
   PUT_LINE("  end ADVANCE;");
   NEW_LINE;
end  EMIT_ADVANCE_TLR;


procedure  EMIT_PKG_DECLS  is
   --  Emit the declarations for the generated package body.
begin
   NEW_LINE;
   PUT_LINE("  BUFFER_SIZE: constant := 100;");
   PUT_LINE("  subtype BUFFER_INDEX is INTEGER range 1..BUFFER_SIZE;");
   NEW_LINE;
   --  Emit an enumerated type definition for the defined pattern names.
   PUT_LINE("  type PATTERN_ID is (");
   EMITTED_CHARS := 22;
   for  I in 1 .. CUR_TABLE_ENTRIES  loop
      EMIT_PATTERN_NAME(PATTERN_TABLE(I).NAME);
```

```
      PUT(',');
      EMITTED_CHARS := EMITTED_CHARS + 1;
   end loop;
   if EMITTED_CHARS /= 0 then
     NEW_LINE;
   end if;
   PUT_LINE(" END_OF_INPUT, END_OF_LINE, UNRECOGNIZED);");
   NEW_LINE;
   --  Emit the package variable declarations
   PUT_LINE("  CUR_LINE_NUM: NATURAL := 0;");
   PUT_LINE("  CUR_PATTERN: PATTERN_ID := END_OF_LINE;");
   PUT_LINE("  START_OF_LINE: BOOLEAN;");
   PUT_LINE("  CHAR_BUFFER: STRING(BUFFER_INDEX);");
   PUT_LINE("  CUR_CHAR_NDX: BUFFER_INDEX;");
   PUT_LINE("  TOP_CHAR_NDX: BUFFER_INDEX;");
   NEW_LINE;
   --  Emit the fixed procedure definitions
   PUT_LINE("  procedure SCAN_PATTERN;  -- forward");
   NEW_LINE;
   PUT_LINE("  function CURRENT_SYMBOL return STRING is");
   PUT_LINE("  begin");
   PUT_LINE("    return CHAR_BUFFER(1..(CUR_CHAR_NDX-1));");
   PUT_LINE("  end;");
end  EMIT_PKG_DECLS;


procedure  EMIT_PATTERN_NAME ( FILE: in FILE_TYPE; NAME: in LLSTRINGS )  is
   -- Write the name of a pattern to a specified file.
begin
   for  I in LLSTRINGS'RANGE  loop
   exit when  NAME(I) = ' ';
      PUT( FILE, NAME(I) );
   end loop;
end  EMIT_PATTERN_NAME;


procedure  EMIT_PATTERN_NAME ( NAME: in LLSTRINGS )  is
   -- Write the name of a pattern to the standard output file.
```

```
begin
   for  I in LLSTRINGS'RANGE  loop
   exit when  NAME(I) = ' ';
      PUT( NAME(I) );
      EMITTED_CHARS := EMITTED_CHARS + 1;
   end loop;
   if  EMITTED_CHARS > OUTPUT_LINE_LIMIT  then
      NEW_LINE;
      EMITTED_CHARS := 0;
   end if;
end  EMIT_PATTERN_NAME;


procedure  EMIT_SCAN_PROC  is

   --  Generate the pattern-matching code for referenced patterns.

   procedure  EMIT_SELECT ( SEL_SET: in SELECTION_SET );
      --  Generate an expression for the selection set SEL_SET.

   procedure  EMIT_PATTERN_MATCH ( PAT: in out LLATTRIBUTE;
                                   NAME: in LLSTRINGS;
                                   SHOW_NAME: in BOOLEAN;
                                   PARENT_NULLABLE: in BOOLEAN;
                                   LOOK_AHEAD: BOOLEAN )  is

      --  Generate pattern-matching code from a normal-form pattern.

      procedure  EMIT_ALT_CASES( INIT_PAT: in out LLATTRIBUTE;
                                 PARENT_NULLABLE: in BOOLEAN )  is
         --  Generate "when" clauses for an alternation pattern.
         PAT: LLATTRIBUTE := INIT_PAT;
      begin
         while  PAT.VARIANT = ALT  loop
            -- emit successive alternatives
            PUT("  when ");
            EMIT_SELECT(PAT.LEFT.SEL_SET);
            PUT_LINE(" =>");
```

```
      if NAME = ANONYMOUS then
         EMIT_PATTERN_MATCH( PAT.LEFT, PAT.LEFT.NAME, SHOW_NAME,
            PARENT_NULLABLE, LOOK_AHEAD );
      else
         EMIT_PATTERN_MATCH( PAT.LEFT, NAME, SHOW_NAME,
            PARENT_NULLABLE, LOOK_AHEAD );
      end if;
      INIT_PAT.COULD_FAIL := INIT_PAT.COULD_FAIL or
                             PAT.LEFT.COULD_FAIL;
      PAT := PAT.RIGHT;
   end loop;
   -- emit the last alternative
   PUT("  when ");
   EMIT_SELECT(PAT.SEL_SET);
   PUT_LINE(" =>");
   if NAME = ANONYMOUS then
      EMIT_PATTERN_MATCH( PAT, PAT.NAME, SHOW_NAME,
         PARENT_NULLABLE, LOOK_AHEAD );
   else
      EMIT_PATTERN_MATCH( PAT, NAME, SHOW_NAME,
         PARENT_NULLABLE, LOOK_AHEAD );
   end if;
   INIT_PAT.COULD_FAIL := INIT_PAT.COULD_FAIL or PAT.COULD_FAIL;
end  EMIT_ALT_CASES;

procedure  EMIT_CONCAT_RIGHT( SHOW_NAME: in BOOLEAN )  is

   --  Emit the right-hand part of a concatenation pattern.

   procedure  EMIT_CONCAT_CASES  is
   begin
      case PAT.RIGHT.VARIANT is
         when  ALT | LOOK | OPT | REP  =>
            EMIT_PATTERN_MATCH( PAT.RIGHT, ANONYMOUS, SHOW_NAME,
               PARENT_NULLABLE, LOOK_AHEAD );
         when  CAT | RNG  =>
            PUT_LINE("case CURRENT_CHAR is");
            PUT("  when ");
```

```
            EMIT_SELECT(PAT.RIGHT.SEL_SET);
            PUT_LINE(" =>");
            if NAME = ANONYMOUS then
               EMIT_PATTERN_MATCH( PAT.RIGHT, PAT.RIGHT.NAME,
                  SHOW_NAME, PARENT_NULLABLE,
                  LOOK_AHEAD and PAT.RIGHT.VARIANT = CAT );
            else
               EMIT_PATTERN_MATCH( PAT.RIGHT, NAME, SHOW_NAME,
                  PARENT_NULLABLE,
                  LOOK_AHEAD and PAT.RIGHT.VARIANT = CAT );
            end if;
            PUT_LINE("  when others =>");
            if PARENT_NULLABLE then
               PUT_LINE("     CUR_CHAR_NDX := FALL_BACK_NDX;");
               PUT_LINE("     LOOK_AHEAD_FAILED := TRUE;");
            else
               PUT_LINE("     CUR_PATTERN := UNRECOGNIZED;");
            end if;
            PUT_LINE("end case;");
            PAT.RIGHT.COULD_FAIL := TRUE;
         when others  =>
            --  No other kinds of patterns should show up here.
            PUT_LINE("CUR_PATTERN := UNRECOGNIZED;");
            PAT.RIGHT.COULD_FAIL := TRUE;
      end case;
   end  EMIT_CONCAT_CASES;

begin   -- EMIT_CONCAT_RIGHT(SHOW_NAME)
   --  Emit the right-hand part of a concatenation pattern.
   if  PAT.LEFT.COULD_FAIL  then
      if  PARENT_NULLABLE  then
         PUT_LINE("if not LOOK_AHEAD_FAILED then");
      else
         PUT_LINE("if CUR_PATTERN /= UNRECOGNIZED then");
      end if;
      EMIT_CONCAT_CASES;
      PUT_LINE("end if;");
   else
```

```
         EMIT_CONCAT_CASES;
      end if;
   end  EMIT_CONCAT_RIGHT;

begin   --  EMIT_PATTERN_MATCH(PAT,NAME,SHOW_NAME,
        --     PARENT_NULLABLE,LOOK_AHEAD)
   --  Generate pattern-matching code from a normal-form pattern.
   case PAT.VARIANT is
      when  ALT  =>
         PUT_LINE("case CURRENT_CHAR is");
         EMIT_ALT_CASES( PAT, PAT.NULLABLE or PARENT_NULLABLE );
         PUT("  when others =>");
         if  PAT.NULLABLE  then
            PUT_LINE(" null;");
            PUT_LINE("end case;");
            if SHOW_NAME and NAME /= ANONYMOUS then
               PUT("CUR_PATTERN := ");
               EMIT_PATTERN_NAME(NAME);
               PUT_LINE(";");
            end if;
         else
            NEW_LINE;
            if PARENT_NULLABLE then
               PUT_LINE("    CUR_CHAR_NDX := FALL_BACK_NDX;");
               PUT_LINE("    LOOK_AHEAD_FAILED := TRUE;");
            else
               PUT_LINE("    CUR_PATTERN := UNRECOGNIZED;");
            end if;
            PAT.COULD_FAIL := TRUE;
            PUT_LINE("end case;");
         end if;
      when  CAT  =>
         if PAT.RIGHT.NULLABLE then
            if NAME = ANONYMOUS then
               EMIT_PATTERN_MATCH(PAT.LEFT,PAT.LEFT.NAME,SHOW_NAME,
                  PARENT_NULLABLE,LOOK_AHEAD);
               EMIT_CONCAT_RIGHT(SHOW_NAME);
            else
```

```
            EMIT_PATTERN_MATCH(PAT.LEFT,NAME,SHOW_NAME,
               PARENT_NULLABLE,LOOK_AHEAD);
            EMIT_CONCAT_RIGHT(FALSE);
         end if;
      else
         EMIT_PATTERN_MATCH(PAT.LEFT,ANONYMOUS,FALSE,
            PARENT_NULLABLE,PARENT_NULLABLE);
         EMIT_CONCAT_RIGHT(SHOW_NAME);
      end if;
      PAT.COULD_FAIL := PAT.LEFT.COULD_FAIL or PAT.RIGHT.COULD_FAIL;
   when  LOOK  =>
      PUT_LINE("case CURRENT_CHAR is");
      PUT("  when");
      EMIT_SELECT(PAT.LEFT.SEL_SET);
      PUT_LINE(" =>");
      PUT_LINE("    LOOK_AHEAD_NDX := CUR_CHAR_NDX;");
      EMIT_PATTERN_MATCH(PAT.LEFT,PAT.LEFT.NAME,
         SHOW_NAME,TRUE,FALSE);
      PUT_LINE("  when others =>");
      PUT_LINE("    LOOK_AHEAD_FAILED := TRUE;");
      PUT_LINE("end case;");
      PUT_LINE("CUR_CHAR_NDX := LOOK_AHEAD_NDX;");
      PUT_LINE("if LOOK_AHEAD_FAILED then");
      PUT_LINE("  case CURRENT_CHAR is");
      PUT_LINE("    when");
      EMIT_SELECT(PAT.LEFT.SEL_SET);
      PUT_LINE(" =>");
      EMIT_PATTERN_MATCH(PAT.RIGHT,PAT.RIGHT.NAME,
         SHOW_NAME,FALSE,FALSE);
      PUT_LINE("    when others =>");
      if PAT.RIGHT.NULLABLE then
         PUT_LINE("      null;");
      else
         PUT_LINE("      CUR_PATTERN := UNRECOGNIZED;");
      end if;
      PUT_LINE("  end case;");
      PUT_LINE("end if;");
      PAT.COULD_FAIL := PAT.RIGHT.COULD_FAIL;
```

```
when  OPT  =>
   PUT_LINE("case CURRENT_CHAR is");
   PUT("  when ");
   EMIT_SELECT(PAT.SEL_SET);
   PUT_LINE(" =>");
   if NAME = ANONYMOUS then
      EMIT_PATTERN_MATCH(PAT.EXPR,PAT.NAME,
         SHOW_NAME,TRUE,LOOK_AHEAD);
   else
      EMIT_PATTERN_MATCH(PAT.EXPR,NAME,
         SHOW_NAME,TRUE,LOOK_AHEAD);
   end if;
   PUT_LINE("  when others => null;");
   PUT_LINE("end case;");
   PAT.COULD_FAIL := PAT.EXPR.COULD_FAIL;
when  REP  =>
   PUT_LINE("loop");
   PUT_LINE("  case CURRENT_CHAR is");
   if  PAT.EXPR.VARIANT = ALT  then
      EMIT_ALT_CASES(PAT.EXPR,TRUE);
   else
      PUT("    when ");
      EMIT_SELECT(PAT.SEL_SET);
      PUT_LINE(" =>");
      EMIT_PATTERN_MATCH(PAT.EXPR,NAME,SHOW_NAME,TRUE,LOOK_AHEAD);
   end if;
   PUT_LINE("    when others => exit;");
   PUT_LINE("  end case;");
   if  PAT.EXPR.COULD_FAIL  then
      PUT_LINE("exit when LOOK_AHEAD_FAILED;");
      PAT.COULD_FAIL := TRUE;
   end if;
   PUT_LINE("end loop;");
when  RNG  =>
   if LOOK_AHEAD then
     PUT_LINE("LOOK_AHEAD;");
   else
     PUT_LINE("CHAR_ADVANCE;");
```

```
            end if;
            if SHOW_NAME and NAME /= ANONYMOUS then
               PUT("CUR_PATTERN := ");
               EMIT_PATTERN_NAME(NAME);
               PUT_LINE(";");
            end if;
         when others  =>
            --  No other kinds of patterns should show up here.
            if PARENT_NULLABLE then
               PUT_LINE("LOOK_AHEAD_FAILED := TRUE;");
            end if;
            PUT_LINE("CUR_PATTERN := UNRECOGNIZED;");
            PAT.COULD_FAIL := TRUE;
      end case;
   end  EMIT_PATTERN_MATCH;

   procedure  EMIT_SELECT ( SEL_SET: in SELECTION_SET )  is
      --  Generate an expression for the selection set SEL_SET.
      STATE: INTEGER range 0..3 := 0;

      procedure  EMIT_CHAR( CH: in CHARACTER )  is
      begin
         case CH is
            when ASCII.BEL =>
               PUT("ASCII.BEL");
            when ASCII.BS =>
               PUT("ASCII.BS");
            when ASCII.HT =>
               PUT("ASCII.HT");
            when ASCII.LF =>
               PUT("ASCII.LF");
            when ASCII.VT =>
               PUT("ASCII.VT");
            when ASCII.FF =>
               PUT("ASCII.FF");
            when ASCII.CR =>
               PUT("ASCII.CR");
            when ASCII.ESC =>
```

```
            PUT("ASCII.ESC");
         when ' '..'~' =>
            PUT('''); PUT(CH); PUT(''');
         when ASCII.DEL =>
            PUT("ASCII.DEL");
         when others =>
            PUT("ASCII.NUL");
      end case;
   end;

begin
   for  CH in SELECTION_SET'RANGE   loop
      case  STATE  is
         when  0  =>
            --  Initial state, looking for selection set characters.
            if SEL_SET(CH) then
               EMIT_CHAR(CH);
               STATE := 1;
            end if;
         when  1  =>
            --  Have produced first character, is it a range?
            if SEL_SET(CH) then
               PUT("..");
               STATE := 2;
            else
               STATE := 3;
            end if;
         when  2  =>
            --  Have produced first char and "..", looking for end char.
            if not SEL_SET(CH) then
               EMIT_CHAR(CHARACTER'PRED(CH));
               STATE := 3;
            end if;
         when  3  =>
            --  Have produced one or more alt. terms, looking for more.
            if SEL_SET(CH) then
               PUT(" | ");
               EMIT_CHAR(CH);
```

```
                  STATE := 1;
               end if;
            end case;
         end loop;
         --  Check for a possible loose end.
         if STATE = 2 then
            EMIT_CHAR(SEL_SET'LAST);
         end if;
      end  EMIT_SELECT;

   begin   --  EMIT_SCAN_PROC
      NEW_LINE;
      --  Generate the pattern-matching code for referenced patterns.
      PUT_LINE("  procedure SCAN_PATTERN is");
      NEW_LINE;
      PUT_LINE("    CURRENT_CHAR: CHARACTER;");
      PUT_LINE("    END_OF_INPUT_STREAM: BOOLEAN;");
      PUT_LINE("    LOOK_AHEAD_FAILED: BOOLEAN := FALSE;");
      PUT_LINE("    FALL_BACK_NDX: BUFFER_INDEX := 1;");
      PUT_LINE("    LOOK_AHEAD_NDX: BUFFER_INDEX;");
      NEW_LINE;
      PUT_LINE("    procedure CHAR_ADVANCE is");
      PUT_LINE("    begin");
      PUT_LINE("      CUR_CHAR_NDX := CUR_CHAR_NDX+1;");
      PUT_LINE("      FALL_BACK_NDX := CUR_CHAR_NDX;");
      PUT_LINE("      if CUR_CHAR_NDX <= TOP_CHAR_NDX then");
                        -- take the next character from the buffer
      PUT_LINE("        CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);");
      PUT_LINE("      else");
                        -- fetch the next character and put it in the buffer
      PUT_LINE("        GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);");
      PUT_LINE("        if END_OF_INPUT_STREAM then");
      PUT_LINE("          CURRENT_CHAR := ASCII.etx;");
      PUT_LINE("        end if;");
      PUT_LINE("        CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;");
      PUT_LINE("        TOP_CHAR_NDX := CUR_CHAR_NDX;");
      PUT_LINE("      end if;");
      PUT_LINE("    end;");
```

```
NEW_LINE;
PUT_LINE("    procedure LOOK_AHEAD is");
PUT_LINE("    begin");
PUT_LINE("      CUR_CHAR_NDX := CUR_CHAR_NDX+1;");
PUT_LINE("      if CUR_CHAR_NDX <= TOP_CHAR_NDX then");
                  -- take the next character from the buffer
PUT_LINE("        CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);");
PUT_LINE("      else");
                  -- fetch the next character and put it in the buffer
PUT_LINE("        GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);");
PUT_LINE("        if END_OF_INPUT_STREAM then");
PUT_LINE("          CURRENT_CHAR := ASCII.etx;");
PUT_LINE("        end if;");
PUT_LINE("        CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;");
PUT_LINE("        TOP_CHAR_NDX := CUR_CHAR_NDX;");
PUT_LINE("      end if;");
PUT_LINE("    end;");
NEW_LINE;
PUT_LINE("  begin");
if  LEXICON = null  then
   PUT_LINE(STANDARD_ERROR,
            "*** No patterns were referenced for code generation.");
   PUT_LINE("    CUR_PATTERN := UNRECOGNIZED;");
else
   PUT_LINE("    START_OF_LINE := CUR_PATTERN = END_OF_LINE;");
   PUT_LINE("    if START_OF_LINE then");
   PUT_LINE("      CUR_LINE_NUM := CUR_LINE_NUM+1;");
   PUT_LINE("      TOP_CHAR_NDX := 1;");
   PUT_LINE("      GET_CHARACTER(END_OF_INPUT_STREAM,CHAR_BUFFER(1));");
   PUT_LINE("      if END_OF_INPUT_STREAM then");
   PUT_LINE("        CHAR_BUFFER(1) := ASCII.etx;");
   PUT_LINE("      end if;");
   PUT_LINE("    else");
                  -- shift the buffer contents forward
   PUT_LINE("      TOP_CHAR_NDX := TOP_CHAR_NDX-CUR_CHAR_NDX+1;");
   PUT_LINE("      for N in 1..TOP_CHAR_NDX loop");
   PUT_LINE("        CHAR_BUFFER(N) := CHAR_BUFFER(N+CUR_CHAR_NDX-1);");
   PUT_LINE("      end loop;");
```

```
      PUT_LINE("    end if;");
      PUT_LINE("    CUR_CHAR_NDX := 1;");
      PUT_LINE("    CURRENT_CHAR := CHAR_BUFFER(1);");
      PUT_LINE("    case CURRENT_CHAR is");
      PUT_LINE("      when ASCII.etx =>");
      PUT_LINE("        CUR_PATTERN := END_OF_INPUT;");
      PUT_LINE("      when ASCII.lf..ASCII.cr =>");
      PUT_LINE("        CUR_PATTERN := END_OF_LINE;");
      while  LEXICON.VARIANT = ALT  loop
         --  Emit successive alternatives.
         PUT("      when ");
         EMIT_SELECT(LEXICON.LEFT.SEL_SET);
         PUT_LINE(" =>");
         it LEXICON.NAME = ANONYMOUS then
            EMIT_PATTERN_MATCH(LEXICON.LEFT,LEXICON.LEFT.NAME,
               TRUE,FALSE,FALSE);
         else
            EMIT_PATTERN_MATCH(LEXICON.LEFT,LEXICON.NAME,
               TRUE,FALSE,FALSE);
            LEXICON.RIGHT.NAME := LEXICON.NAME;
         end if;
         LEXICON := LEXICON.RIGHT;
      end loop;
      --  Emit the last alternative.
      PUT("      when ");
      EMIT_SELECT(LEXICON.SEL_SET);
      PUT_LINE(" =>");
      EMIT_PATTERN_MATCH(LEXICON,LEXICON.NAME,TRUE,FALSE,FALSE);
      PUT_LINE("      when others =>");
      PUT_LINE("        CHAR_ADVANCE;");
      PUT_LINE("        CUR_PATTERN := UNRECOGNIZED;");
      PUT_LINE("    end case;");
   end if;
   PUT_LINE("  end;");
   NEW_LINE;
end  EMIT_SCAN_PROC;
```

```
procedure  EMIT_TOKEN( TOKEN: in LLATTRIBUTE )  is
   --  Emit an identifier or literal token value.
begin
   case  TOKEN.VARIANT  is
      when  CHAR  =>
         PUT(''');
         PUT(TOKEN.CHAR_VAL);
         PUT(''');
         EMITTED_CHARS := EMITTED_CHARS + 3;
      when  IDENT | LIT  =>
         if  TOKEN.VARIANT = IDENT  then
            --  Precede it with a blank.
            PUT(' ');
            EMITTED_CHARS := EMITTED_CHARS + 1;
         elsif  TOKEN.STRING_VAL(1) = ';'  then
            PUT_LINE(";");
            EMITTED_CHARS := 0;
            return;
         end if;
         for  I in LLSTRINGS'RANGE  loop
         exit when  TOKEN.STRING_VAL(I) = ' ';
            PUT( TOKEN.STRING_VAL(I) );
            EMITTED_CHARS := EMITTED_CHARS + 1;
         end loop;
      when  STR  =>
         PUT('"');
         EMITTED_CHARS := EMITTED_CHARS + 1;
         for  I in 2 .. LLSTRINGS'LAST  loop
            PUT(TOKEN.STRING_VAL(I));
            EMITTED_CHARS := EMITTED_CHARS + 1;
         exit when  TOKEN.STRING_VAL(I) = '"';
         end loop;
      when others  =>
         --  No other kinds of patterns should show up here.
         null;
   end case;
   if  EMITTED_CHARS > OUTPUT_LINE_LIMIT  then
      NEW_LINE;
```

```
      EMITTED_CHARS := 0;
   end if;
end  EMIT_TOKEN;


procedure  INCLUDE_PATTERN( PAT_ID: in LLATTRIBUTE )  is
   --  Include a referenced pattern for code generation.
   --  Global variable LEXICON holds the complete definition of all
   --  patterns encountered so far in the actions part of a lexical
   --  analyzer specification.
   N: INTEGER range 0 .. PATTERN_TABLE_SIZE;
begin
   N := LOOK_UP_PATTERN(PAT_ID);
   if  N = 0  then
      PUT(STANDARD_ERROR,"*** Pattern """);
      EMIT_PATTERN_NAME(STANDARD_ERROR,PAT_ID.STRING_VAL);
      PUT(STANDARD_ERROR,""" called for in line ");
      PUT(STANDARD_ERROR,0,1);
      PUT_LINE(STANDARD_ERROR," is not defined.");
   else
      LEXICON := ALTERNATE(PATTERN_TABLE(N),LEXICON);
      COMPLETE_PAT(LEXICON);
   end if;
end  INCLUDE_PATTERN;


function  LOOK_AHEAD ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE  is
   --  Create a look-ahead pattern.
begin
   return  new TREE_NODE'(LOOK,ANONYMOUS,(others => FALSE),
                          FALSE,FALSE,PAT,BAD_PATTERN);
end  LOOK_AHEAD;


function  LOOK_UP_PATTERN ( PAT_ID: in LLATTRIBUTE ) return  INTEGER  is
   --  Return the index of the named pattern in the pattern table.
begin
   for  I in 1 .. CUR_TABLE_ENTRIES  loop
```

```
      if  PAT_ID.STRING_VAL = PATTERN_TABLE(I).NAME  then
         --  You found it.
         return  I;
      end if;
   end loop;
   --  If the name is not in the table then
   return  0;
end  LOOK_UP_PATTERN;


function  OPTION ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE  is
   --  Form an optional pattern.
begin
   case  PAT.VARIANT  is
      when  ALT | CAT | IDENT | RNG  =>
         return  new TREE_NODE'(OPT,ANONYMOUS,(others => FALSE),
                                TRUE,FALSE,PAT);
      when  OPT | REP  =>
         --  Just copy the original node.
         return  new TREE_NODE'(PAT.all);
      when others  =>
         --  No other kinds of patterns should show up here.
         return BAD_PATTERN;
   end case;
end  OPTION;


function  REPEAT ( PAT: in LLATTRIBUTE ) return  LLATTRIBUTE  is
   --  Form a repetition pattern.
begin
   case  PAT.VARIANT  is
      when  ALT | CAT | IDENT | RNG  =>
         return  new TREE_NODE'(REP,ANONYMOUS,(others => FALSE),
                                TRUE,FALSE,PAT);
      when  OPT | REP  =>
         --  Simplify the repeated pattern.
         return  new TREE_NODE'(REP,ANONYMOUS,(others => FALSE),
                                TRUE,FALSE,PAT.EXPR);
```

```
      when others  =>
         --  No other kinds of patterns should show up here.
         return BAD_PATTERN;
   end case;
end  REPEAT;


procedure  STORE_PATTERN ( PAT_ID, PAT: in LLATTRIBUTE )  is
   --  Store a pattern definition in the pattern table.
   --  Patterns are stored in alphabetical order by name.
begin
   if  CUR_TABLE_ENTRIES = PATTERN_TABLE_SIZE  then
      --  I guess I didn't make the table big enough.
      raise PATTERN_TABLE_FULL;
   end if;
   for  I in 1 .. CUR_TABLE_ENTRIES  loop
      if  PAT_ID.STRING_VAL < PATTERN_TABLE(I).NAME  then
         -- Insert the name here.
         for  K in reverse I .. CUR_TABLE_ENTRIES  loop
            PATTERN_TABLE(K+1) := PATTERN_TABLE(K);
         end loop;
         PATTERN_TABLE(I) := PAT;
         PATTERN_TABLE(I).NAME := PAT_ID.STRING_VAL;
         CUR_TABLE_ENTRIES := CUR_TABLE_ENTRIES + 1;
         return;
      elsif  PAT_ID.STRING_VAL = PATTERN_TABLE(I).NAME  then
         --  Combine this definition with the previous one(s).
         PATTERN_TABLE(I) := ALTERNATE( PAT, PATTERN_TABLE(I) );
         PATTERN_TABLE(I).NAME := PAT_ID.STRING_VAL;
         return;
      end if;
   end loop;
   CUR_TABLE_ENTRIES := CUR_TABLE_ENTRIES + 1;
   PATTERN_TABLE(CUR_TABLE_ENTRIES) := PAT;
   PATTERN_TABLE(CUR_TABLE_ENTRIES).NAME := PAT_ID.STRING_VAL;
end  STORE_PATTERN;

end  LL_SUPPORT;
```

APPENDIX C

Lexical Analyzer Test Data

This appendix contains listings of programs and data used to test the lexical analyzer generator. The first listing is the test driver program used to exercise generated code. Following this are three test cases. Test #1 is a simple test of the code-generation templates. Test #2 exercises the generator's handling of look-ahead and conversion of patterns into canonical form. Test #3 is a test of the lexical analyzer used to replace the generator's bootstrap analyzer. The analyzer specification and generated code for Test #3 are given in Appendix A. Input for Test #3 was its own specification from Appendix A.

Contents:

```
with  INTEGER_TEXT_IO, TEXT_IO;

procedure TEST_DRIVER is

   --  This procedure is a simple test program for exercising code
   --  produced by the Lexical Analyzer Generator.

   use INTEGER_TEXT_IO, TEXT_IO;

   type TOKEN_TYPE is
           (ALT, CAT, CHAR, DOTS, IDENT, KEY, LIT, NOT_MINE,
            NUMBER, OPERATOR, OPT, REP, RNG, SPECIAL, STR);

   subtype SHORT_STRING is STRING(1..12);

   type TOKEN is
      record
         KIND: TOKEN_TYPE;
         PRINTVALUE: SHORT_STRING;
         LINENUMBER: INTEGER;
      end record;

   EOS: BOOLEAN;
   TOK: TOKEN;

   procedure GET_CHARACTER( EOS: out BOOLEAN;
                 NEXT: out CHARACTER;
                 MORE: in BOOLEAN := TRUE ) is
   --  Produce input characters for the lexical analyzer.
   begin
      if  END_OF_FILE(STANDARD_INPUT)  then
         EOS := TRUE;
      elsif  END_OF_LINE(STANDARD_INPUT)  then
         SKIP_LINE(STANDARD_INPUT);
         EOS := FALSE;
         NEXT := ASCII.CR;
      else
         EOS := FALSE;
```

```
      GET(STANDARD_INPUT,NEXT);
   end if;
end;


function  MAKE_TOKEN(KIND: TOKEN_TYPE; SYMBOL: STRING; LINENUMBER: NATURAL )
      return  TOKEN  is

--  construct a token value from input lexical information

   function CVT_STRING( STR: in STRING ) return SHORT_STRING is
   --  Convert an arbitrary-length string to a fixed length string.
      RESULT: SHORT_STRING;
   begin
      for  I in SHORT_STRING'RANGE  loop
         if  I <= STR'LAST  then
            RESULT(I) := STR(I);
         else
            RESULT(I) := ' ';
         end if;
      end loop;
      return RESULT;
   end;

begin
   return  TOKEN'(KIND, CVT_STRING(SYMBOL), LINENUMBER);
end;


package TOKEN_STREAM is

  procedure ADVANCE(EOS: out BOOLEAN;
                    NEXT: out TOKEN;
                    MORE: in BOOLEAN := TRUE);

end TOKEN_STREAM;

package body TOKEN_STREAM is separate;
```

```
begin
   loop
      TOKEN_STREAM.ADVANCE(EOS,TOK);
   exit when EOS;
      PUT(TOK.PRINTVALUE);
      PUT("   ");
      PUT(TOK.LINENUMBER);
      PUT("   ");
      case TOK.KIND is
         when ALT       => PUT("Alternation");
         when CAT       => PUT("Concatenation");
         when CHAR      => PUT("Character");
         when DOTS      => PUT("Dots");
         when IDENT     => PUT("Identifier");
         when KEY       => PUT("Keyword");
         when LIT       => PUT("Literal");
         when NOT_MINE  => PUT("Unrecognized");
         when NUMBER    => PUT("Number");
         when OPERATOR  => PUT("Operator");
         when OPT       => PUT("Option");
         when REP       => PUT("Repetition");
         when RNG       => PUT("Range");
         when SPECIAL   => PUT("Special Symbol");
         when STR       => PUT("String");
      end case;
      NEW_LINE;
   end loop;
end TEST_DRIVER;
```

```
separate ( TEST_DRIVER )

lexicon TOKEN_STREAM is

-- The following patterns test the lexical analyzer
-- generator's basic code generation templates.

patterns

   Alternate  ::=  Letter | Digit ;

   Concat  ::=  '&' Digit ;

   Digit  ::=  '0'..'9' ;

   Letter  ::=  'A'..'Z' | 'a'..'z' ;

   Option  ::=  '~' [ Digit ] ;

   Repetition  ::=  '*' { Digit } ;

actions

   when Alternate =>
      NEXT := MAKE_TOKEN(ALT,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Concat =>
      NEXT := MAKE_TOKEN(CAT,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Option =>
      NEXT := MAKE_TOKEN(OPT,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Repetition =>
      NEXT := MAKE_TOKEN(REP,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;
```

```
    when others =>
       NEXT := MAKE_TOKEN(NOT_MINE,CURRENT_SYMBOL,CUR_LINE_NUM);
       return;

end TOKEN_STREAM;
```

```
separate ( TEST_DRIVER )

package body TOKEN_STREAM is

  BUFFER_SIZE: constant := 100;
  subtype BUFFER_INDEX is INTEGER range 1..BUFFER_SIZE;

  type PATTERN_ID is
    (Alternate,Concat,Digit,Letter,Option,Repetition,
     END_OF_INPUT, END_OF_LINE, UNRECOGNIZED);

  CUR_LINE_NUM: NATURAL := 0;
  CUR_PATTERN: PATTERN_ID := END_OF_LINE;
  START_OF_LINE: BOOLEAN;
  CHAR_BUFFER: STRING(BUFFER_INDEX);
  CUR_CHAR_NDX: BUFFER_INDEX;
  TOP_CHAR_NDX: BUFFER_INDEX;

  procedure SCAN_PATTERN;  -- forward

  function CURRENT_SYMBOL return STRING is
  begin
    return CHAR_BUFFER(1..(CUR_CHAR_NDX-1));
  end;

  procedure ADVANCE(EOS: out BOOLEAN;
    NEXT: out TOKEN;
    MORE: in BOOLEAN := TRUE) is
  begin
    EOS := FALSE;
    loop
      SCAN_PATTERN;
      case CUR_PATTERN is
        when END_OF_INPUT =>
          EOS := TRUE;
          return;
        when END_OF_LINE => null;
        when Alternate =>
```

```
        NEXT:= MAKE_TOKEN( ALT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Concat =>
        NEXT:= MAKE_TOKEN( CAT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Option =>
        NEXT:= MAKE_TOKEN( OPT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Repetition =>
        NEXT:= MAKE_TOKEN( REP, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when others =>
        NEXT:= MAKE_TOKEN( NOT_MINE, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
    end case;
  end loop;
end ADVANCE;


procedure SCAN_PATTERN is

  CURRENT_CHAR: CHARACTER;
  END_OF_INPUT_STREAM: BOOLEAN;
  LOOK_AHEAD_FAILED: BOOLEAN := FALSE;
  FALL_BACK_NDX: BUFFER_INDEX := 1;
  LOOK_AHEAD_NDX: BUFFER_INDEX;

  procedure CHAR_ADVANCE is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    FALL_BACK_NDX := CUR_CHAR_NDX;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
```

```
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

  procedure LOOK_AHEAD is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

begin
  START_OF_LINE := CUR_PATTERN = END_OF_LINE;
  if START_OF_LINE then
    CUR_LINE_NUM := CUR_LINE_NUM+1;
    TOP_CHAR_NDX := 1;
    GET_CHARACTER(END_OF_INPUT_STREAM,CHAR_BUFFER(1));
    if END_OF_INPUT_STREAM then
      CHAR_BUFFER(1) := ASCII.etx;
    end if;
  else
    TOP_CHAR_NDX := TOP_CHAR_NDX-CUR_CHAR_NDX+1;
    for N in 1..TOP_CHAR_NDX loop
      CHAR_BUFFER(N) := CHAR_BUFFER(N+CUR_CHAR_NDX-1);
    end loop;
  end if;
  CUR_CHAR_NDX := 1;
  CURRENT_CHAR := CHAR_BUFFER(1);
  case CURRENT_CHAR is
```

```
when ASCII.etx =>
  CUR_PATTERN := END_OF_INPUT;
when ASCII.lf..ASCII.cr =>
  CUR_PATTERN := END_OF_LINE;
when '*' =>                              -- Code for repetition pattern
  CHAR_ADVANCE;
  CUR_PATTERN := Repetition;
  loop
    case CURRENT_CHAR is
      when '0'..'9' =>
        CHAR_ADVANCE;
      when others => exit;
    end case;
  end loop;
when '~' =>                              -- Code for option pattern
  CHAR_ADVANCE;
  CUR_PATTERN := Option;
  case CURRENT_CHAR is
    when '0'..'9' =>
      CHAR_ADVANCE;
    when others => null;
  end case;
when '&' =>                              -- Code for concatenation pattern
  CHAR_ADVANCE;
  case CURRENT_CHAR is
    when '0'..'9' =>
      CHAR_ADVANCE;
      CUR_PATTERN := Concat;
    when others =>
      CUR_PATTERN := UNRECOGNIZED;
  end case;
when 'A'..'Z' | 'a'..'z' =>              -- Code for alternation and range
  CHAR_ADVANCE;
  CUR_PATTERN := Alternate;
when '0'..'9' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Alternate;
when others =>
```

```
        CHAR_ADVANCE;
        CUR_PATTERN := UNRECOGNIZED;
    end case;
  end;

end TOKEN_STREAM;
```

INPUT:

```
A&1~*
1&2~3*4567890
abc&1&2&3~4~~5**6*7890
/&/~/*/
```

OUTPUT:

A	1	Alternation
&1	1	Concatenation
~	1	Option
*	1	Repetition
1	2	Alternation
&2	2	Concatenation
~3	2	Option
*4567890	2	Repetition
a	3	Alternation
b	3	Alternation
c	3	Alternation
&1	3	Concatenation
&2	3	Concatenation
&3	3	Concatenation
~4	3	Option
~	3	Option
~5	3	Option
*	3	Repetition
*6	3	Repetition
*7890	3	Repetition
/	4	Unrecognized
&	4	Unrecognized

/	4	Unrecognized
~	4	Option
/	4	Unrecognized
*	4	Repetition
/	4	Unrecognized

```
separate ( TEST_DRIVER )

lexicon TOKEN_STREAM is

-- The following specification tests the lexical analyzer
-- generator's handling of look-ahead, conversion of patterns
-- into canonical form, and pattern simplifications.

patterns

   Digit  ::=  '0'..'9' ;

   Dots  ::=  '.' | ".." ;

   Identifier  ::=  'A'..'Z' [ Digit ] ;

   Keyword  ::=  "FOR" | "GO" | "IF" | "LET" | "NEXT" ;

   Number  ::=  Digit { ['_'] Digit } ;

actions

   when Dots =>
      NEXT := MAKE_TOKEN(DOTS,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Identifier =>
      NEXT := MAKE_TOKEN(IDENT,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Keyword =>
      NEXT := MAKE_TOKEN(KEY,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;

   when Number =>
      NEXT := MAKE_TOKEN(NUMBER,CURRENT_SYMBOL,CUR_LINE_NUM);
      return;
```

```
    when others =>
       NEXT := MAKE_TOKEN(NOT_MINE,CURRENT_SYMBOL,CUR_LINE_NUM);
       return;

end TOKEN_STREAM;
```

```
separate ( TEST_DRIVER )

package body TOKEN_STREAM is

  BUFFER_SIZE: constant := 100;
  subtype BUFFER_INDEX is INTEGER range 1..BUFFER_SIZE;

  type PATTERN_ID is
    (Digit,Dots,Identifier,Keyword,Number,
     END_OF_INPUT, END_OF_LINE, UNRECOGNIZED);

  CUR_LINE_NUM: NATURAL := 0;
  CUR_PATTERN: PATTERN_ID := END_OF_LINE;
  START_OF_LINE: BOOLEAN;
  CHAR_BUFFER: STRING(BUFFER_INDEX);
  CUR_CHAR_NDX: BUFFER_INDEX;
  TOP_CHAR_NDX: BUFFER_INDEX;

  procedure SCAN_PATTERN;  -- forward

  function CURRENT_SYMBOL return STRING is
  begin
    return CHAR_BUFFER(1..(CUR_CHAR_NDX-1));
  end;

  procedure ADVANCE(EOS: out BOOLEAN;
    NEXT: out TOKEN;
    MORE: in BOOLEAN := TRUE) is
  begin
    EOS := FALSE;
    loop
      SCAN_PATTERN;
      case CUR_PATTERN is
        when END_OF_INPUT =>
          EOS := TRUE;
          return;
        when END_OF_LINE => null;
        when Dots =>
```

```
        NEXT:= MAKE_TOKEN( DOTS, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Identifier =>
        NEXT:= MAKE_TOKEN( IDENT, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Keyword =>
        NEXT:= MAKE_TOKEN( KEY, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when Number =>
        NEXT:= MAKE_TOKEN( NUMBER, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
      when others =>
        NEXT:= MAKE_TOKEN( NOT_MINE, CURRENT_SYMBOL, CUR_LINE_NUM);
        return;
    end case;
  end loop;
end ADVANCE;


procedure SCAN_PATTERN is

  CURRENT_CHAR: CHARACTER;
  END_OF_INPUT_STREAM: BOOLEAN;
  LOOK_AHEAD_FAILED: BOOLEAN := FALSE;
  FALL_BACK_NDX: BUFFER_INDEX := 1;
  LOOK_AHEAD_NDX: BUFFER_INDEX;

  procedure CHAR_ADVANCE is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    FALL_BACK_NDX := CUR_CHAR_NDX;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
```

```
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

  procedure LOOK_AHEAD is
  begin
    CUR_CHAR_NDX := CUR_CHAR_NDX+1;
    if CUR_CHAR_NDX <= TOP_CHAR_NDX then
      CURRENT_CHAR := CHAR_BUFFER(CUR_CHAR_NDX);
    else
      GET_CHARACTER(END_OF_INPUT_STREAM,CURRENT_CHAR);
      if END_OF_INPUT_STREAM then
        CURRENT_CHAR := ASCII.etx;
      end if;
      CHAR_BUFFER(CUR_CHAR_NDX) := CURRENT_CHAR;
      TOP_CHAR_NDX := CUR_CHAR_NDX;
    end if;
  end;

begin
  START_OF_LINE := CUR_PATTERN = END_OF_LINE;
  if START_OF_LINE then
    CUR_LINE_NUM := CUR_LINE_NUM+1;
    TOP_CHAR_NDX := 1;
    GET_CHARACTER(END_OF_INPUT_STREAM,CHAR_BUFFER(1));
    if END_OF_INPUT_STREAM then
      CHAR_BUFFER(1) := ASCII.etx;
    end if;
  else
    TOP_CHAR_NDX := TOP_CHAR_NDX-CUR_CHAR_NDX+1;
    for N in 1..TOP_CHAR_NDX loop
      CHAR_BUFFER(N) := CHAR_BUFFER(N+CUR_CHAR_NDX-1);
    end loop;
  end if;
  CUR_CHAR_NDX := 1;
  CURRENT_CHAR := CHAR_BUFFER(1);
  case CURRENT_CHAR is
```

```
when ASCII.etx =>
  CUR_PATTERN := END_OF_INPUT;
when ASCII.lf..ASCII.cr =>
  CUR_PATTERN := END_OF_LINE;
when '0'..'9' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Number;
  loop
    case CURRENT_CHAR is
      when '0'..'9' =>                 -- Code for numbers
        CHAR_ADVANCE;
      when '_' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when '0'..'9' =>
            CHAR_ADVANCE;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when others => exit;
    end case;
  exit when LOOK_AHEAD_FAILED;
  end loop;
when 'A'..'E' | 'H' | 'J'..'K' | 'M' | 'O'..'Z' =>
  CHAR_ADVANCE;
  CUR_PATTERN := Identifier;           -- Code for identifiers
  case CURRENT_CHAR is
    when '0'..'9' =>
      CHAR_ADVANCE;
    when others => null;
  end case;
when '.' =>                            -- Code for dots
  CHAR_ADVANCE;
  CUR_PATTERN := Dots;
  case CURRENT_CHAR is
    when '.' =>
      CHAR_ADVANCE;
```

```
      when others => null;
    end case;
  when 'F' =>                           -- Code for keyword "FOR"
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier;
    case CURRENT_CHAR is
      when '0'..'9' =>
        CHAR_ADVANCE;
      when 'O' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when 'R' =>
            CHAR_ADVANCE;
            CUR_PATTERN := Keyword;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when others => null;
    end case;
  when 'G' =>                           -- Code for keyword "GO"
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier;
    case CURRENT_CHAR is
      when 'O' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Keyword;
      when '0'..'9' =>
        CHAR_ADVANCE;
      when others => null;
    end case;
  when 'I' =>                           -- Code for keyword "IF"
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier ;
    case CURRENT_CHAR is
      when 'F' =>
        CHAR_ADVANCE;
        CUR_PATTERN := Keyword;
```

```
      when '0'..'9' =>
        CHAR_ADVANCE;
      when others => null;
    end case;
  when 'N' =>                              -- Code for keyword "NEXT"
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier;
    case CURRENT_CHAR is
      when 'E' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
          when 'X' =>
            LOOK_AHEAD;
            case CURRENT_CHAR is
              when 'T' =>
                CHAR_ADVANCE;
                CUR_PATTERN := Keyword;
              when others =>
                CUR_CHAR_NDX := FALL_BACK_NDX;
                LOOK_AHEAD_FAILED := TRUE;
            end case;
          when others =>
            CUR_CHAR_NDX := FALL_BACK_NDX;
            LOOK_AHEAD_FAILED := TRUE;
        end case;
      when '0'..'9' =>
        CHAR_ADVANCE;
      when others => null;
    end case;
  when 'L' =>                              -- Code for keyword "LET"
    CHAR_ADVANCE;
    CUR_PATTERN := Identifier;
    case CURRENT_CHAR is
      when '0'..'9' =>
        CHAR_ADVANCE;
      when 'E' =>
        LOOK_AHEAD;
        case CURRENT_CHAR is
```

```
              when 'T' =>
                CHAR_ADVANCE;
                CUR_PATTERN := Keyword;
              when others =>
                CUR_CHAR_NDX := FALL_BACK_NDX;
                LOOK_AHEAD_FAILED := TRUE;
            end case;
          when others => null;
        end case;
      when others =>
        CHAR_ADVANCE;
        CUR_PATTERN := UNRECOGNIZED;
    end case;
  end;

end TOKEN_STREAM;
```

INPUT:

```
.AFOR1..Z8IF123
A1B2C3.....123 456 789
FORGOIFLETNEXT
FO1G2I3LE4NEX5
```

OUTPUT:

.	1	Dots
A	1	Identifier
FOR	1	Keyword
1	1	Number
..	1	Dots
Z8	1	Identifier
IF	1	Keyword
123	1	Number
A1	2	Identifier
B2	2	Identifier
C3	2	Identifier
..	2	Dots
..	2	Dots
.	2	Dots
123	2	Number
	2	Unrecognized
456	2	Number
	2	Unrecognized
789	2	Number
FOR	3	Keyword
GO	3	Keyword
IF	3	Keyword

LET	3	Keyword
NEXT	3	Keyword
F	4	Identifier
O1	4	Identifier
G2	4	Identifier
I3	4	Identifier
L	4	Identifier
E4	4	Identifier
N	4	Identifier
E	4	Identifier
X5	4	Identifier

```
separate          2   Identifier
(                 2   Literal
LL_COMPILE        2   Identifier
)                 2   Literal
lexicon           4   Identifier
LL_TOKENS         4   Identifier
is                4   Identifier
patterns          9   Identifier
Graphic_Char     11   Identifier
::=              11   Literal
..               11   Literal
;                11   Literal
Letter           13   Identifier
::=              13   Literal
..               13   Literal
|                13   Literal
..               13   Literal
;                13   Literal
Digit            15   Identifier
::=              15   Literal
..               15   Literal
;                15   Literal
Letter_or_Di     17   Identifier
::=              17   Literal
Letter           17   Identifier
|                17   Literal
Digit            17   Identifier
;                17   Literal
Character_Li     19   Identifier
::=              19   Literal
Graphic_Char     19   Identifier
;                19   Literal
Comment          21   Identifier
::=              21   Literal
"--"             21   String
[                21   Literal
Graphic_Char     21   Identifier
]                21   Literal
```

```
;                  21  Literal
Delimiter          23  Identifier
::=                23  Literal
|                  23  Literal
|                  23  Literal
|                  23  Literal
"**"               23  String
|                  23  Literal
|                  24  Literal
|                  24  Literal
|                  24  Literal
|                  24  Literal
"/="               24  String
|                  24  Literal
|                  25  Literal
":="               25  String
|                  25  Literal
|                  25  Literal
"<<"               25  String
|                  25  Literal
"<="               25  String
|                  25  Literal
"<>"               25  String
|                  26  Literal
|                  26  Literal
|                  26  Literal
">="               26  String
|                  26  Literal
">>"               26  String
;                  26  Literal
Identifier         28  Identifier
::=                28  Literal
Letter             28  Identifier
{                  28  Literal
[                  28  Literal
]                  28  Literal
Letter_or_Di       28  Identifier
}                  28  Literal
```

;	28	Literal
Number	30	Identifier
::=	30	Literal
Digit	30	Identifier
{	30	Literal
[	30	Literal
]	30	Literal
Digit	30	Identifier
}	30	Literal
;	30	Literal
Special_Symb	32	Identifier
::=	32	Literal
|	32	Literal
|	32	Literal
|	32	Literal
|	32	Literal
".."	32	String
|	33	Literal
"::="	33	String
|	33	Literal
|	33	Literal
"=>"	33	String
|	33	Literal
|	33	Literal
|	34	Literal
|	34	Literal
|	34	Literal
;	34	Literal
String_Liter	36	Identifier
::=	36	Literal
Quoted_Strin	36	Identifier
{	36	Literal
Quoted_Strin	36	Identifier
}	36	Literal
;	36	Literal
Quoted_Strin	38	Identifier
::=	38	Literal
[	38	Literal

Non_Quote_Ch	38	Identifier
}	38	Literal
;	38	Literal
Non_Quote_Ch	40	Identifier
::=	40	Literal
´..	40	Literal
|	40	Literal
..	40	Literal
;	40	Literal
White_Space	42	Identifier
::=	42	Literal
Separator	42	Identifier
{	42	Literal
Separator	42	Identifier
}	42	Literal
;	42	Literal
Separator	44	Identifier
::=	44	Literal
|	44	Literal
ASCII	44	Identifier
.	44	Literal
HT	44	Identifier
;	44	Literal
actions	46	Identifier
when	48	Identifier
Character_Li	48	Identifier
=>	48	Literal
NEXT	49	Identifier
:=	49	Literal
MAKE_TOKEN	49	Identifier
(	49	Literal
CHAR	49	Identifier
,	49	Literal
CURRENT_SYMB	49	Identifier
,	49	Literal
CUR_LINE_NUM	49	Identifier
)	49	Literal
;	49	Literal

return	50	Identifier
;	50	Literal
when	52	Identifier
Comment	52	Identifier
|	52	Literal
White_Space	52	Identifier
=>	52	Literal
null	52	Identifier
;	52	Literal
when	54	Identifier
Delimiter	54	Identifier
|	54	Literal
Number	54	Identifier
|	54	Literal
Special_Symb	54	Identifier
=>	54	Literal
NEXT	55	Identifier
:=	55	Literal
MAKE_TOKEN	55	Identifier
(	55	Literal
LIT	55	Identifier
,	55	Literal
CURRENT_SYMB	55	Identifier
,	55	Literal
CUR_LINE_NUM	55	Identifier
)	55	Literal
;	55	Literal
return	56	Identifier
;	56	Literal
when	58	Identifier
Identifier	58	Identifier
=>	58	Literal
NEXT	59	Identifier
:=	59	Literal
MAKE_TOKEN	59	Identifier
(	59	Literal
IDENT	59	Identifier
,	59	Literal

CURRENT_SYMB	59	Identifier
,	59	Literal
CUR_LINE_NUM	59	Identifier
)	59	Literal
;	59	Literal
return	60	Identifier
;	60	Literal
when	62	Identifier
String_Liter	62	Identifier
=>	62	Literal
NEXT	63	Identifier
:=	63	Literal
MAKE_TOKEN	63	Identifier
(	63	Literal
STR	63	Identifier
,	63	Literal
CURRENT_SYMB	63	Identifier
,	63	Literal
CUR_LINE_NUM	63	Identifier
)	63	Literal
;	63	Literal
return	64	Identifier
;	64	Literal
when	66	Identifier
others	66	Identifier
=>	66	Literal
NEXT	67	Identifier
:=	67	Literal
MAKE_TOKEN	67	Identifier
(	67	Literal
LIT	67	Identifier
,	67	Literal
CURRENT_SYMB	67	Identifier
,	67	Literal
CUR_LINE_NUM	67	Identifier
)	67	Literal
;	67	Literal
return	68	Identifier

;	68	Literal
end	70	Identifier
LL_TOKENS	70	Identifier
;	70	Literal

Distribution List for IDA Paper P-2108

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John F. Kramer Program Manager STARS DARPA/ISTO 1400 Wilson Blvd. Arlington, VA 22209-2308	4
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Katydean Price, CSED	1
Dr. Reginald N. Meeson, CSED	1
IDA Control & Distribution Vault	2